Spring 2015

SemCache: Semantics-Aware Caching for Efficient GPU Offloading

Nabeel Al-Saber
Purdue University

Graduate School Form 30
(Updated 11/20/2014)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Nabeel Al-Saber

Entitled
SemCache: Semantics-Aware Caching for Efficient GPU Offloading

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

MILIND KULKARNI

ARUN PRAKASH

SAMUEL P. MIDKIFF

VIJAY S. PAI

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

MILIND KULKARNI

Approved by Major Professor(s): _____

_____

Approved by: Michael R. Melloch 03/02/2015

Head of the Department Graduate Program Date

SEMCACHE: SEMANTICS-AWARE CACHING FOR EFFICIENT GPU OFFLOADING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Nabeel Al-Saber

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2015

Purdue University

West Lafayette, Indiana

This thesis is dedicated to my loving parents, my wife and my son. Thank you for providing me with constant support and inspiration.

ACKNOWLEDGMENTS

I would like to thank The Almighty, for all the blessings throughout my life.

Special thanks to my Advisor Professor Milind Kulkarni for his encouragement, supervision and support through the Ph.D. program. I would like to thank my advisory committee: Professor Samuel Midkiff, Professor Arun Prakash and Professor Vijay S. Pai for their feedback and suggestions during this work.

This research is supported by the Department of Energy under contract DE-FC02-12ER26104. The GPU hardware we used was provided by an equipment grant from Nvidia.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Al-Saber, Nabeel Ph.D., Purdue University, May 2015. SemCache: Semantics-Aware Caching for Efficient GPU Offloading. Major Professor: Milind Kulkarni.

Graphical Processing Units (GPUs) offer massive, highly-efficient parallelism, making them an attractive target for computation-intensive applications. However, GPUs have a separate memory space which introduces the complexity of manually handling explicit data movements between GPU and CPU memory spaces. Although GPU kernels/libraries have made it easy to improve application performance by offloading computation to GPUs, unfortunately it is very difficult to manually optimize CPU-GPU communication between multiple kernel invocations to avoid redundant communication when using these kernels with complex applications.

In this thesis, we introduce SemCache [1], a semantics-aware GPU cache that automatically manages CPU-GPU communication in addition to optimizing communication by eliminating redundant transfers using caching. It uses library semantics to determine the appropriate caching granularity for a given offloaded library (*e.g.*, matrices). Our caching technique is efficient; it only tracks matrices instead of tracking every memory access at fine granularity. We applied SemCache to Basic Linear Algebra Subprograms (BLAS) [2] library to provide a GPU drop-in replacement library.

SemCache++ [3] extends SemCache to support offloading to multiple GPUs. SemCache++ is used to build the first multi-GPU drop-in replacement library that (a) uses the virtual memory to automatically manage and optimize multi-GPU communication and (b) requires no program rewriting or annotations. SemCache++ also enables new features like asynchronous transfers, parallel execution and overlapping communication with computation.

Experimental results show that our system can dramatically reduce redundant communication for real-world computational science application and deliver significant performance improvements, beating GPU-based implementations like MAGMA [4], CULA [5], CUBLAS and CUBLASXT [6].

1. INTRODUCTION

Graphics processing units (GPUs) offer massive, highly-efficient parallelism, making them an attractive target for computation-intensive applications. Due to the difficulty of programming GPUs, a practical option for leveraging their capabilities is to *offload* computation using libraries. For example, there are many GPU implementations of linear algebra libraries [4–7], which outperform CPU implementations of popular libraries such as BLAS [2] and LAPACK [8] by taking advantage of the GPU's parallel hardware. Such GPU libraries allow existing applications written against the BLAS and LAPACK APIs to easily benefit from execution on heterogeneous platforms: most computation executes on the CPU, but invocations of BLAS methods are executed on the GPU.

1.1 Single GPU Offloading

This library-based offloading approach to harnessing the power of GPUs has some drawbacks. Notably, moving data back and forth between the CPU and the GPU incurs significant expense, making optimizing this communication paramount when library calls are composed. If successive library calls operate on the same data, the data should be moved to the GPU just once, rather than separately for each call, while data should only be transferred back to the CPU if a computation requires it. Such optimization is in tension with the encapsulation objectives of library-based offloading: if a programmer has to manually manage communication between the CPU and GPU, she can no longer port her program to a heterogeneous system without modification.

Consider the simple case of a series of matrix multiply operations, as shown in Figure 1.1. Each matrix multiply requires that the source matrices be on the GPU

```
1 GEMM(A, B, C); //C = A * B
2 ...
3 GEMM(B, C, D); //D = B * C
4 ...
5 GEMM(C, D, E); //E = C * D
```

Fig. 1.1. Simple example with repeated matrix multiplication

and the result matrix be transferred back to the CPU. However, the naïve approach of transferring the sources to the GPU and the results back on every operation results in redundant communication. Some source matrices (*e.g.*, `B`) are transferred to the GPU twice, while other matrices (*e.g.*, `C` in the second operation) are transferred to the GPU even though they were computed on the GPU originally. A better approach is to transfer `B` just once, and use it for both operations, while consuming `C` directly from the GPU for the second operation.

There exist libraries of GPU kernels (*e.g.*, CULA Standard Interface) that attempt to ease the process of offloading computation to the GPU by automatically handling data movement and execution on the GPU. Because these libraries target specific operations (*e.g.*, linear algebra), using them is often as simple as replacing operations in an application with equivalent GPU versions; in fact, because computational applications are often already implemented with libraries such as BLAS [2] and LAPACK [8], CULA implementations of those operations can be used with essentially no modifications to program code and no need for GPU expertise. Unfortunately, such drop-in replacement libraries come with drawbacks. The libraries do not consider the composition of library calls, instead implementing each offloaded operation as a self-contained unit. As a result, the libraries do not consider the possibility of redundant data movement across operations (in other words, they adopt the naïve communication approach described above). *Because each operation is offloaded in isolation, the composition of operations may not be implemented efficiently.*

To correctly minimize data movement and avoid redundancy when offloading computation to the GPU, the composition of offloaded operations must be considered. While lower-level libraries (such as CUBLAS, CULA's "device interface," or MAGMA [4]) give the programmer precise control over data movement (so that, *e.g.*, he can avoid transferring matrix `B` to the GPU twice in the previous example), it is often difficult to reason about which data movement might be redundant and which might be necessary. This is especially true in large, modular applications, where operations might be quite distant from one another both in the code and during execution, and where a single piece of static code may exhibit data redundancy based entirely on when and where the code is invoked during execution (consider a method called from several places in an application that performs several linear algebra operations on matrices passed in as parameters). In such a scenario, any attempt to statically determine whether communication is necessary is doomed to failure; *simply providing low-level control of data movement is not enough to allow eliminating redundant communication.*

What is needed is an *automatic* approach to managing data movement between GPU and CPU that can *dynamically* determine whether data movement is necessary and hence provide drop-in replacements for computational libraries. Such an approach will allow programmers to achieve efficient communication for heterogeneous computing without adopting a new programming model.

1.2 Multi-GPU Offloading

In recent years, *Multi-GPU* systems are becoming increasingly popular, with multiple GPUs available for computation offloading. Unfortunately, handling multi-GPU systems is substantially harder than managing a single GPU, as now computation and data need to be distributed across multiple GPUs. To simplify multi-GPU offloading, libraries such as CUBLASXT [6], MAGMA [4], CULA [5] and FLAME [7], completely encapsulate communication in their library calls: prior to invoking a method, data

is transferred to the GPU(s), and upon completion, data is transferred back. Such encapsulation introduces significant overheads, as much of this data movement is redundant. However, without encapsulation, managing data movement between kernels is quite difficult in multi-GPU systems.

While there have been several attempts at developing multi-GPU frameworks that can optimize communication more thoroughly, they are not well-suited to developing library replacements. They either require adopting a new programming model [9, 10] or require annotating every CPU data access, including those outside the offloaded library call [11]. The burden of rewriting an application or annotating large numbers of data accesses makes these models hard to adopt for large applications. What is needed is a framework for developing GPU libraries with the *appearance* of fully-encapsulated library calls, but the *performance* of more tightly coupled interaction between the CPU and GPU.

1.3 Contributions

In this thesis, we propose a *semantics-aware GPU cache* to reduce redundant communication between the CPU and the GPU. At a high level, our software caching approach treats the CPU and GPU memory spaces as two caches, and uses an MSI (modified/shared/invalid) protocol to maintain coherence between them. When a method is called to execute on the GPU, the cache state of the data used by the method is inspected, and data is transferred to the GPU only if it does not already reside there. When data is modified on the CPU, the cache is used to invalidate any corresponding data on the GPU.

Crucially, the cache we develop is *generic*: the system itself is not tied to any particular library. Instead, all of the semantic information is provided in the library implementation, allowing the same caching system to be reused for different libraries, in each case providing tuned cache implementations that use the correct granularity for a given library.

This thesis makes the following contributions:

- The design and implementation of SemCache, a generic GPU cache that automatically manages CPU-GPU communication and dynamically optimizes communication. It is augmented with semantic information to provide *tuned, library-specific caching.*

- A generalization of this caching solution (akin to memoization) that creates *semantic links* between data on the CPU and GPU, allowing SemCache to automatically eliminate redundant computation and translate between different layouts.

- An annotated GPU BLAS library that provides a *drop in replacement* for existing BLAS libraries that, in conjunction with SemCache, delivers optimized communication between the CPU and GPU.

- Experimental results showing, both for microbenchmarks and a large, real-world computational science application, that SemCache can dramatically reduce redundant communication, and deliver significant performance improvements, beating not only CPU implementations but also GPU-based implementations using existing, tuned libraries.

- An integration between Trilinos [12] and SemCache to enable CPU-GPU automatic and optimized memory management.

SemCache++ extends SemCache to support multiple GPUs with the following contributions:

- The design and implementation of SemCache++, a generic multi-GPU cache that automatically manages communication between CPU and multiple GPUs at variable granularity. SemCache++ enables multi-GPU caching to avoid communication.

- SemCache++ exploits all devices (CPUs and GPUs) in parallel, and uses CUDA streams to allow overlapping of communication and computation.
- A SemCache++-enabled multi-GPU BLAS library that provides a drop-in replacement for existing BLAS libraries.
- Experimental results showing that SemCache++ can dramatically reduce redundant communication, and deliver significant performance improvements over CUBLASXT, NVIDIA's tuned multi-GPU BLAS library.

2. BACKGROUND

2.1 GP-GPU Computing

Over the past few years, Graphics Processing Units (GPUs) have become attractive platforms for computing. The programmable vector units on GPUs offer the potential for massive, energy efficient parallelism. A GPU is generally composed of hundreds of small cores grouped in highly parallel and highly multithreaded streaming multiprocessors (SMs), a high speed interconnection network, and a device memory (global memory) as shown in Figure 2.1.

Today, it is very common for phones, desktops, laptops, clusters, supercomputers, and cloud environments to include both CPUs and GPUs. Originally, GPUs where

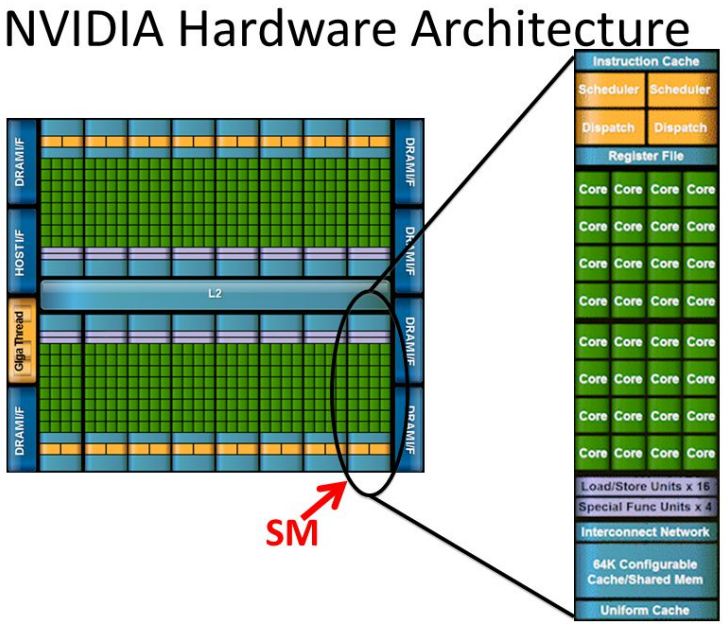

Fig. 2.1. NVIDIA's GPU Architecture [13]

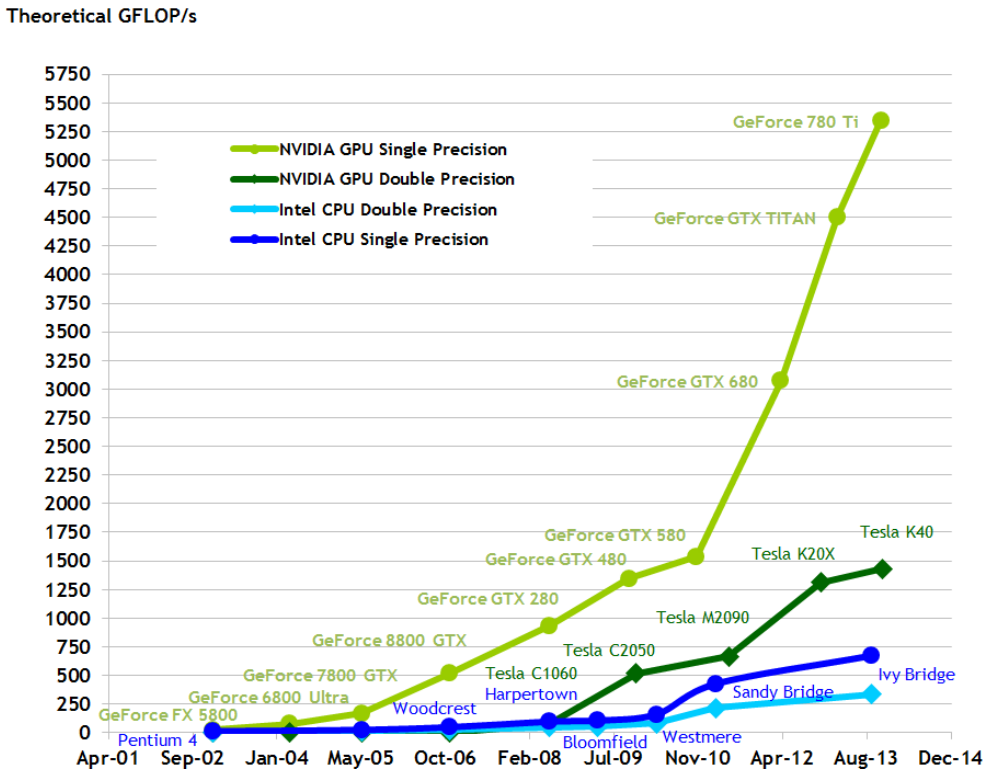

Fig. 2.2. Floating-Point Operations per Second for the CPU and GPU [13]

intended to speedup graphics processing but today they are also used in general processing (GP) to accelerate data-parallel computations in scientific and engineering applications. Figure 2.2 shows how the GPU performance outperforms the CPU.

There are two main downsides, to GP-GPU computing. First, to achieve their energy efficiency, GPU cores are very simple, and only provide performance benefits when executing carefully parallelized code. Hence, attempting to port general code to GPUs is a tedious task, and often results in ineffective code. Instead, it is more effective to execute only those portions of an application that are amenable to GPU-style parallelism, such as linear algebra code, on the GPU, leaving the remainder of the application code on the CPU as shown in Figure 2.3. Because writing efficient implementations on a GPU is difficult even for algorithms well-suited to parallel execution, there has been a proliferation of libraries that provide GPU implementations

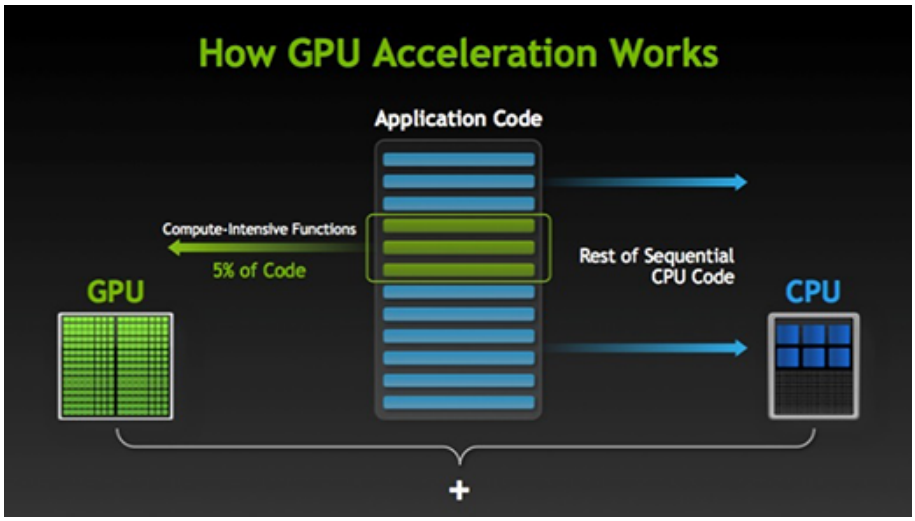

Fig. 2.3. GPU Offloading [13]

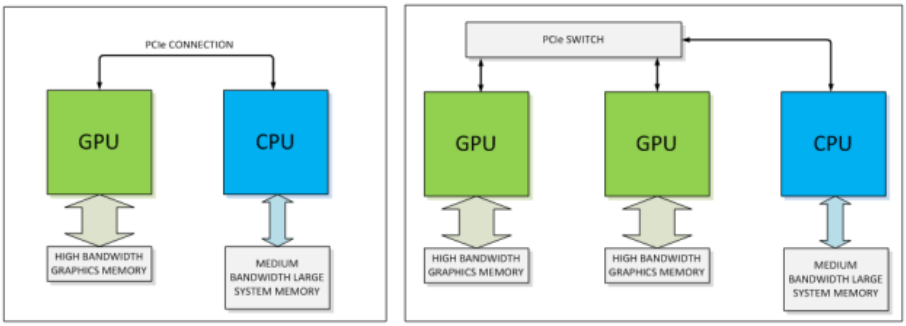

Fig. 2.4. GPU connected to CPU using PCIe [13]

of common linear algebra kernels (often providing the BLAS interface [2]), easing the task of offloading these operations to the GPU.

The second downside to using GPUs for general purpose computing is that most GPUs use separate memory from the CPU. In other words, the GPU uses a separate address space from the CPU, and hence the two processing units cannot readily share data. Instead, data must be explicitly transferred between the CPU and the GPU. This limitation is especially problematic when only portions of a computation are offloaded to the GPU: because both the CPU and the GPU perform operations on the same data, the data must be transferred back and forth as necessary. Worse,

transferring data between CPU and GPU is *slow*, especially in comparison to the speed of each processing unit's own memory. The GPU is connected to the CPU using the PCI express bus, Figure 2.4. The maximum theoretical bandwidth for PCIe V2 is 8GB/s. Performing data transfers are often a significant cost of GPU computation, and there have been several approaches that have attempted to avoid even offloading computation when data transfer costs exceed the benefit of GPU computation [14–16].

2.2 Offloading Libraries to GPUs

The necessity for explicit data movement between GPU and CPU makes providing modular libraries that provide GPU kernels more difficult. Consider the example code in Figure 1.1. As discussed in the introduction, there are several distinct linear algebra operations performed in this example, each of which would be performed by a different library call. In the interests of modularity and encapsulation, some libraries handle communication between the CPU and GPU "under-the-hood" like CULA standard interface. While this makes using the library easier, it results in redundant communication. The library calls are implemented to execute in isolation, and as self-contained units, they assume that the data resides on the CPU. When invoking a method, a library call must (i) allocate space for the arguments and result on the GPU; (ii) transfer the arguments from the CPU to the GPU; (iii) perform the computation on the GPU; and (iv) transfer the result back to the CPU. As a result, even if multiple library calls could make use of the same data, new space is allocated and the data is transferred for each call. Hence, as we see in Figure 2.5(a), at each call two matrices are transferred to the GPU and one is transferred back, for a total of 9 matrix transfers.

Clearly, full encapsulation introduces too many performance problems. Instead, other library approaches, such as CUBLAS [6], give the programmer control over data allocation and movement. Hence, as in Figure 2.5(b), the programmer can explicitly

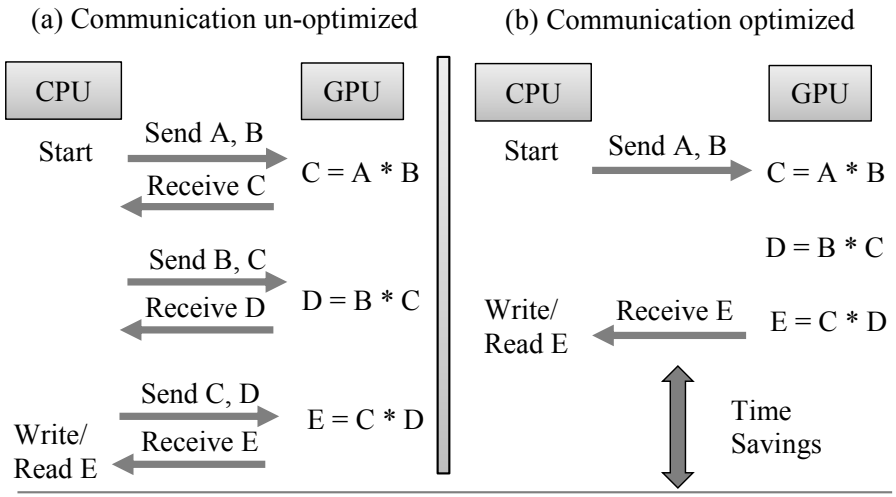

Fig. 2.5. Communication comparison for optimized and un-optimized communication

transfer `A` and `B` to the GPU, allocate space for the results matrices in GPU memory (assuming matrices fit in the GPU memory), and operate only in GPU memory until the final result, `E`, is transferred back to the CPU. This results in the minimal amount of communication, 3 matrix transfers. Unfortunately, forcing a programmer to explicitly manage data requires the programmer to reason about the composition of GPU operations. This is a global task that may be impractical for large codes.

In fact, for highly modular codes, it may not be possible to manually manage data movement. Consider, for example, if the three matrix-multiply calls of our example occurred during different invocations of the same larger method within an application. In other words, the three matrix multiplies occurred from library calls from the same line of code, just with different arguments. Clearly, there is no way to introduce data transfer operations statically to such code to correctly transfer the matrices only when necessary. Whether or not data needs to be transferred to the GPU is a purely *run-time* property, based on what other library methods have been called, and what arguments are being passed to a particular library invocation.

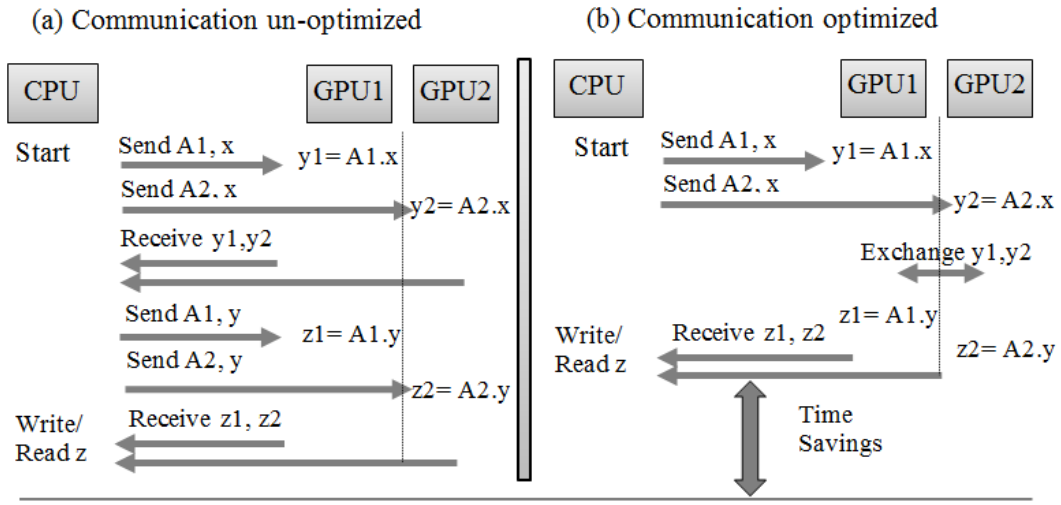

Fig. 2.6. Communication comparison of Encapsulated Multi-GPU libraries and hand tuned communication

2.2.1 Multi-GPU drop-in libraries

The most popular approach to leveraging multiple GPUs is to provide libraries that encapsulate the necessary decomposition and communication. CUBLASXT [17], MAGMA [4] and CULA [5] all provide subsets of BLAS and LAPACK methods that have been optimized for multiple GPUs. As described in the introduction, this encapsulation carries with it a cost: each method call is optimized in isolation, so any opportunities to identify and avoid redundant communication *across* library calls are lost. Essentially, data is "based" at the CPU, and is only distributed among GPUs for the duration of the method call, resulting in redundant communication of shared operands across method calls, and unnecessary communication of result operands when they are not necessary on the CPU.

To understand the difficulty of managing data movement between CPUs and multiple GPUs, consider distributing a series of matrix-vector multiplications (MVMs): $y = A * x; z = A * y$. (This type of computation arises in algorithms such as Jacobi iteration.) To distribute this computation across GPUs, each operation should be decomposed. A natural decomposition is to split A horizontally into two submatrices

A_1 and A_2, sending one to each GPUs. x can then be sent to both GPUs, computing $y_1 = A_1 * x$ and $y_2 = A_2 * x$.

Figure 2.6 demonstrates two different ways that communication could be handled for the remainder of this computation. If the MVM were fully-encapsulated, as in Figure 2.6(a), y_1 and y_2 would be sent back to the CPU and combined into y. When the second MVM is executed, A_1 and A_2 will be re-sent to each GPU, along with the re-composed y. A more efficient approach is to *leave* A in its decomposed form on both GPUs, as in Figure 2.6(b). When the second MVM is invoked, each GPU already has part of y already resident, and need only receive the portion of y they do not already have in order to complete their computation. This dramatically reduces the amount of communication. However, organizing this computation and communication correctly requires realizing that the communication of A is redundant and also that only a portion of y need be communicated. Note that the situation only becomes more complicated if the CPU requires access to the data as well: if code on the CPU (*i.e.*, not in library calls) accesses y between the two MVMs, then y must be fetched back from the GPUs, but the programmer must realize that the portions of y on the GPUs are still valid to avoid performing redundant communication for the second MVM. All in all, efficiently managing communication imposes a significant burden on the programmer.

In libraries such as CUBLASXT, this back-and-forth communication is hidden through pipelining. The computation is broken into chunks, which are distributed among the multiple GPUs. While each GPU is performing a chunk of computation, input data for the next chunk is concurrently sent to the GPU and output data from the previous chunk is retrieved from the GPU. Provided the computation is operating over sufficient data, most of the communication cost can be completely overlapped with computation. Note that the effectiveness of this overlap is dependent on properly choosing the chunk size for pipelining—CUBLASXT leaves the selection of granularity to the programmer, breaking the abstraction layer somewhat.

This pipelining strategy has a deleterious side effect: because the operands must be transferred to the GPU for *every* method call, and the implementation relies on overlapping communication with computation, the communication costs cannot be hidden for small inputs. Moreover, some linear algebra methods, such as SAXPY, simply do not contain enough computation to amortize the communication cost, regardless of how large the input data is (because communication cost grows at the same rate as computation time). Hence, such operations cannot be profitably executed on the GPU, even if there is opportunity to exploit the computation resources of multiple GPUs. As a result, CUBLASXT only provides multi-GPU implementations of BLAS Level 3 methods, while other libraries such as MAGMA also only provide a subset of LAPACK methods. The abstraction boundary imposed by the library interfaces to linear algebra routines precludes exposing communication management to programmers; the only way to support computation offloading efficiently is to automate the data management.

3. RELATED WORK

3.1 GPU Libraries and Programming Models

There are multiple libraries that optimize the performance of Linear Algebra Kernels on GPUs. CUBLAS [6] is an implementation of BLAS [2] (Basic Linear Algebra Subprograms) on top of the NVIDIA's CUDA driver that allows access to the computational resources of NVIDIA GPUs.

MAGMA [4] (Matrix Algebra on GPU and Multicore Architectures) is a heterogeneous algebra library as described on the authors' website:

> MAGMA is a collection of linear algebra libraries for heterogeneous architectures. MAGMA is designed and implemented by the team that developed LAPACK and ScaLAPACK, incorporating the latest developments in hybrid synchronization- and communication-avoiding algorithms, as well as dynamic runtime systems. Interfaces for the current LAPACK and BLAS standards are supported to allow computational scientists to seamlessly port any linear algebra reliant software components to heterogeneous architectures.
>
> MAGMA allows applications to fully exploit the power of current heterogeneous systems of multi/many-core CPUs and multi-GPUs to deliver the fastest possible time to accurate solution within given energy constraints. MAGMA uses a hybridization methodology where algorithms of interest are split into tasks of varying granularity and their execution scheduled over the available hardware components. Scheduling can be static or dynamic. In either case, small non-parallelizable tasks, often on the critical path, are scheduled on the CPU, and larger more parallelizable ones, often Level 3 BLAS, are scheduled on the GPU. [4]

CULA [5] have implemented hybrid GPU accelerated linear algebra routines (LAPACK and BLAS libraries). It provides a standard interface that needs no GPU knowledge in addition to the advanced interface.

These approaches all focus on individual kernels; across kernels, data management must be handled by the programmer.

Other programming models are designed to facilitate heterogeneous scheduling. Intel's Merge [18] is a programming model for heterogeneous multi-core systems. Qilin is a generic programming system that can automatically map computations to GPUs and CPUs through off-line trainings [14]. G-Charm [19] is a runtime system for execution of message-driven parallel applications on hybrid systems. MDR [16] is a performance model-driven runtime for heterogeneous parallel platforms. Such systems try to optimize CPU-GPU communication across the entire program. However, to use them, the programmer must rewrite their application using the specified programming model. In contrast, we are targeting existing large-scale applications, with the goal of optimizing communication without significant programmer effort.

3.2 Automatic Communication Optimization

3.2.1 Compiler based automatic data management

Prior work implemented automatic data management and communication optimization systems for GPUs using compiler analysis with run-time support [20, 21]. Jablin et al. have developed a fully automatic CPU-GPU communication management system (CGCM) [20]. CGCM manages data using a combined run-time and compile-time system without programmer annotations. CGCM requires static analysis (type-inference and alias analysis) because it manages data and optimizes communication at compile-time. The imprecision of static analysis limits CGCM's applicability and performance. Similar to CGCM, AMM [21] uses compiler analysis with run-time support. AMM improves on CGCM compiler analysis to better optimize data communication. The main limitation in these approaches is the use of

static compiler analysis. Such analysis does not scale to large programs because it requires complex inter-procedural analysis and it can not accurately analyze recursive data structures or data-structures with pointer and non-pointer types. In contrast, SemCache uses virtual memory and a more sophisticated run-time system that keeps tracks of data validity status and hence can better optimize communication with less overhead than a static compiler analysis.

3.2.2 DSM based automatic data management

NVIDIA's Unified Memory [13], DyManD [22] and GMAC [23] attempt to manage communication between the GPU and CPU automatically by adopting distributed shared memory (DSM) techniques [24, 25]. These systems use the operating system's page-protection mechanism to detect when data needs to be transferred. Although these techniques are fully automated, they require direct mappings between the CPU and GPU memory spaces. Such single memory space models use the same masked address for data allocated on the CPU and the GPU to simplify address translation. If this direct address mapping is extended to multiple GPUs, each GPU needs to reserve the space for the entire matrix although only a sub-matrix is allocated on each GPU, resulting in wasted GPU memory. As a result, the amount of data shared between the CPU and GPU is limited to the GPU memory size; in fact, the largest inputs that we used in our case study (Section 5.6.3) cannot be handled by existing systems. Furthermore, this direct mapping precludes more complex *semantic* mappings between the CPU and the GPU, such as transforming row-major layout to column-major layout, or SemCache's computation caching (Section 4.4).

The main drawback of CUDA's Unified Memory is transferring matrices at the granularity of pages. In SemCache, if the CPU accesses protected data, it page faults and sends the data back to the CPU from the GPU. Transfers are done at the granularity of matrices. However unlike SemCache, in UM transfers from the GPU to the CPU are done at the granularity of pages. Although this approach

might be efficient for reading small data amounts, it is slow for large data. The high number of page faults and the transfer of small data chunks slows down the transfer process. In libraries like CUBLAS, matrices sizes are usually large and transfers at the granularity of matrices is faster. SemCache uses library semantics to choose the right granularity for data transfers. Recent researchers improved on the performance of unified memory using prefetching and smarter page migration policies [26].

3.3 Multi-GPU Programming Models

Aside from using multi-GPU enabled libraries for offloading, another approach to exploiting multiple GPUs is to use programming models that target general heterogeneous platforms. These approaches tend not to be suitable for library-based offloading, for various reasons. Kim *et al.* develop compiler tools that can automatically distribute an OpenCL kernel across multiple GPUs [27]. However, this work focuses on splitting a single kernel across GPUs, and does not consider how to optimize communication across kernels. MGPU [28] and Trilinos [12] libraries allow the programmer to specify the communication at a high level and the library automatically distributes and executes the workload on multiple GPUs. Unlike SemCache++, such libraries depends on the programmer to manually optimize communication across kernel calls.

VirtCL [29] is a framework for multi-GPU scheduling. It provides an abstraction over OpenCL for scheduling and communication management on multiple GPUs. The main limitation of VirtCL is that it does not automatically split single kernels for execution among several devices. Unlike SemCache++, VirtCL can not speed up a single application unless each kernel is manually split by the programmer.

StarSs [11] is a runtime system to decompose and execute the tasks on multiple GPUs. It requires the programmer to annotate tasks with input/output information in addition to specifying the data movement. StarSs caches data on the GPU to optimize communication across tasks. Unlike drop-in replacement libraries such as those

provided by SemCache++, annotations are prune to errors and they are not enough to automatically manage communication. While StarSs can be used to encapsulate several tasks into library calls, all computation over the data accessed during those calls must be annotated with StarSs directives, *including computations meant to execute on the CPU*. If data is cached on the GPU, any CPU access to the data needs to be annotated to maintain the coherence. Identifying such accesses for annotation is not practical for large-scale applications.

3.3.1 Task Programming Models

StarPU [9] is a task programming library for hybrid architectures. StarPU is described on the authors' website as:

> StarPU's run-time and programming language extensions support a task-based programming model. Applications submit computational tasks, with CPU and/or GPU implementations, and StarPU schedules these tasks and associated data transfers on available CPUs and GPUs. The data that a task manipulates are automatically transferred among accelerators and the main memory, so that programmers are freed from the scheduling issues and technical details associated with these transfers.
>
> StarPU offers a unified offloadable task abstraction named codelet. Rather than rewriting the entire code, programmers can encapsulate existing functions within codelets. In case a codelet can run on heterogeneous architectures, it is possible to specify one function for each architectures (e.g. one function for CUDA and one function for CPUs). StarPU takes care of scheduling and executing those codelets as efficiently as possible over the entire machine, include multiple GPUs. One can even specify several functions for each architecture, and StarPU will automatically determine which version is best for each input size. To relieve programmers from the burden of explicit data transfers, a high-level data management li-

> brary enforces memory coherency over the machine: before a codelet starts (e.g. on an accelerator), all its data are automatically made available on the compute resource. Data are also kept on e.g. GPUs as long as they are needed for further tasks. When a device runs out of memory, StarPU uses an LRU strategy to evict unused data. StarPU also takes care of automatically prefetching data, which thus permits to overlap data transfers with computations (including GPU-GPU direct transfers) to achieve the most of the architecture. [9]

libFLAME [7] is a heterogeneous dense linear algebra library. libFLAME is described on the authors' website as:

> libFLAME is a high performance dense linaer algebra library. The libflame project has developed a runtime system, SuperMatrix, to detect and analyze dependencies found within FLAME algorithms-by-blocks (algorithms whose sub-problems operate only on block operands). The task dependence information is captured in a Directed Acyclic Graph (DAG). Once dependencies are known, the system schedules sub-operations to independent threads of execution. This system is completely abstracted from the algorithm that is being parallelized and requires virtually no change to the algorithm code, but at the same time exposes abundant high-level parallelism. The run-time system uses a software cache to check whether the tiles involved in the operation are already present in-core. Thus, actual data transfers only occur for cache misses. An LRU replacement policy decides which tile is moved back to disk in case there is no place left in the cache to read a new tile, and this is also handled by the run-time. The runtime also supports prefetching and overlapping computation with communication. [7]

PTask [10] is a dataflow programming framework for GPUs that insulates the programmer from low-level details such as device-management, data transfer, and

asynchrony. PTask is supported at the system call interface, so the OS can provide isolation and fairness guarantees for GPU computations. PTask run-time automatically optimize communication and avoids unnecessary data movement.

The fundamental drawback to these approaches is that they require writing the entire program in a task-based programming model. Thus, these models cannot be used to provide library-based offloading, as even the non-library portions of the application must be modified to conform to the model, precluding a "drop-in" replacement for existing linear algebra libraries.

These systems *could* be used to provide GPU replacements for libraries such as BLAS and LAPACK: the rewritten or annotated code could be confined to the library code, providing full encapsulation and optimized communication between multiple routines *inside a single library call*. However, if the same data operated on by the library call is also accessed outside the library, programmers are left with one of two options: (i) rewrite or annotate non-library code, obviating the benefits of library encapsulation; or (ii) give up on communication optimization between library and non-library code, entailing redundant communication. The first option is not compatible with our goal of developing "drop-in" replacement libraries that require minimal program rewriting, while the second option leaves substantial opportunities for optimization on the table.

Table 3.1 shows a comparison between prior multi-GPU libraries and systems and SemCache++. SemCache++ is the only system that optimizations communication without the need to rewrite the program using a different programing model or using annotations for every data access.

Table 3.1
Comparison between Multi-GPU Frameworks

Framework	Program Rewrite/ Annotations	Drop-in	Optimizes comm- unication
SemCache++	x	✓	✓
StarPU	✓	x	✓
PTask	✓	x	✓
StarSs	✓	x	✓
MAGMA	x	✓	x
CUBLASXT	x	✓	x
OpenACC	✓	x	x
OpenCL	✓	x	x
Kokkos	✓	x	x

4. SEMCACHE

This section introduces SemCache [1], a variable-granularity, *Semantics-aware Cache* that can be used to efficiently and easily manage sharing and transferring data between the disjoint CPU and GPU address spaces.

4.1 High Level Overview

A software cache between CPU and GPU, at a high level, is simple and intuitive. One variant, using a MSI (*m*odified, *s*hared, *i*nvalid) protocol might operate as follows: a given piece of memory (*e.g.*, a contiguous block of memory, a page, etc.) is tracked by a run-time system. The run-time tracks whether the contents of the piece of memory are currently valid on both devices (shared), valid only on the GPU (modified on the GPU, invalid on the CPU) or valid only on the CPU (invalid on the GPU, modified on the CPU). Whenever memory is read on a particular device, the cache can be consulted to determine whether the local memory has valid data; if not, communication between GPU and CPU is necessary, and the cache state is changed to shared. When a piece of memory is written on a device, the local cache state is changed to modified, and the state for the other device is changed to invalid.

Such an implementation has been used in numerous previous projects targeting different architectures, from distributed shared memory systems (*e.g.*, [24, 25]) to software caches between Cell processing units (*e.g.*, [30, 31]). The downside to prior implementations is that the granularity at which memory was tracked was constant (*e.g.*, an entire OS page, or a fixed-size block of contiguous memory). However, a fixed granularity may not be appropriate for a given application. If the granularity of the cache is too large (the blocks being tracked are too big), excessive communication will happen, both from transferring unnecessary data and from performing too many

invalidations due to false sharing. If the granularity of the cache is too small, more cache lookups will be necessary for a given set of operations, and communication will be broken up into more transfers, resulting in more overhead. Unfortunately, it is difficult to tell for a given application, what the appropriate cache granularity should be, and different applications may require different granularities.

The key insight of SemCache is that when using libraries to offload computation to GPUs, *the correct granularity for a cache can be inferred.* In particular, the appropriate granularity for the cache should be the data structures operated on during offloaded library calls. Moreover, the library's semantics *directly capture what the relevant data structures are.* As a result, by tying SemCache's granularity to a library's semantics, we can track data at exactly the right granularity for a given application.

For example, when SemCache is used in conjunction with a linear algebra library, the data structures being operated on are matrices; as a result, SemCache will track data at the granularity of the matrices used in a particular application. In contrast, if SemCache is used in conjunction with a graph library, the data structures being operated on might be adjacency lists. SemCache will correctly track data at the granularity of entire adjacency lists representing the graphs being operated on.

SemCache is composed of multiple, interlocking components: (a) a variable-granularity cache structure and interfaces for performing cache lookups, triggering data transfers, and performing invalidations; (b) a strategy for setting the granularity of the cache based on library behavior; and (c) instrumentation and protocols for tracking and maintaining the correct cache state for memory. The following subsections describe these components.

4.2 Cache Design and Structure

The basic design of SemCache is shown in Figure 4.1. There is a single data structure, consisting of a set of *translation records* that tracks the status of the various data structures used in a program. Note that even though data may reside on either the

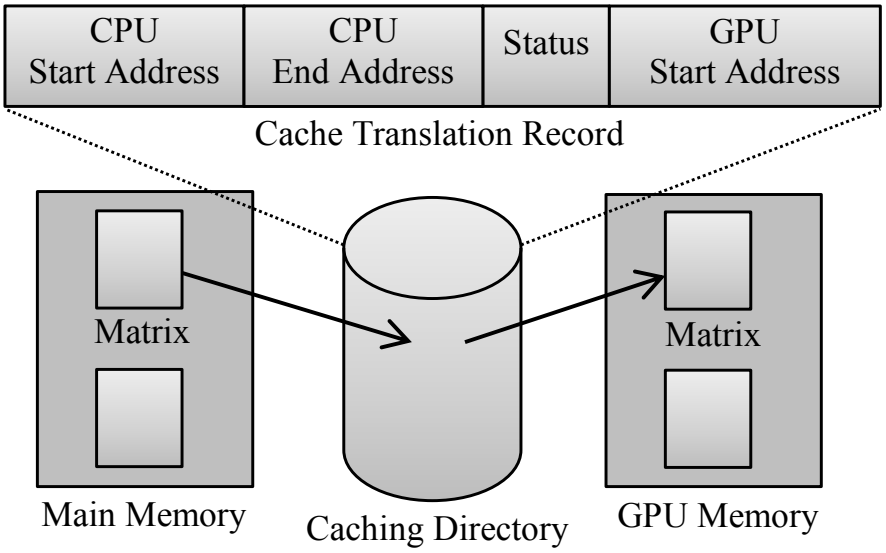

Fig. 4.1. Structures of the Caching Directory

CPU or the GPU, it is the CPU that is in charge of maintaining the cache data, and of performing all lookups and invalidations. This is due to the basic approach used for GPGPU computation. Operations are dispatched to the GPU by transferring data (if necessary) to the GPU and invoking a single kernel method. Once the kernel method completes, control transfers back to the CPU and any necessary data is transferred back. In other words, the CPU alone is responsible for controlling execution and for transferring data between the two memory spaces. As a result, SemCache consists of a single set of translation records maintained by the CPU.

The primary data structure of SemCache is the set of translation records that maintain a mapping between CPU data and the corresponding data on the GPU, as well as the current location of the data. In a sense, SemCache serves as a translation lookaside buffer (TLB), except that its entries point to variable-length regions of memory rather than fixed-size pages. The cache entries are hence indexed by both a start address (cpu_s) and an end address (cpu_e) *of the data's location on the CPU.* Each entry also contains a status field (*status*) to keep track of the data's status. These statuses can be one of C, for valid only on the CPU, G, for valid only on the

GPU, or S, for valid at both locations. Finally, the translation record contains the putative location of the same data on the GPU (gpu_s)[1].

SemCache's interface provides a number of operations. A memory range $[s, e)$ refers to start and end addresses for a memory range on the CPU.

lookup(s, e) Retrieves the translation record associated with memory range $[s, e)$. If the memory range is not currently tracked, create a new entry for the range, and set the status to C.

transferToGPU(entry) Assumes that the status of the entry is S or C. Transfers the contents of memory range $[cpu_s, cpu_e)$ on the CPU to the GPU, allocating new space on the GPU. Sets the GPU start address appropriately. Sets the status of the entry to S.

transferToCPU(entry) Assumes that the status of the entry is S or G. Transfers the contents of memory range $[gpu_s, gpu_s + (cpu_e - cpu_s))$ from the GPU back to the CPU. Sets the status of the entry to S.

invalidateOnGPU(entry) Sets the status of entry to C.

invalidateOnCPU(entry) Sets the status of entry to G.

SemCache maintains the invariant that the ranges tracked by its translation records are disjoint. If a range being looked up is a *subset* of some tracked memory range, then lookup returns the entry associated with the larger memory range. If a range being looked up spans multiple tracked ranges, SemCache*merges all the matching translation records* and creates a new record that tracks a range that spans all of the merged records.

To perform lookups and merges efficiently, SemCache maintains the entries sorted by start address. To look up the range $[s, e)$, SemCache searches for the entry with the largest start address less than or equal to s. If the end address of the found

[1]This location is putative because it is only valid if the status of the range is S or G; if the status is C, the next time the data is sent to the GPU, new space will be allocated for the data

entry is less than or equal to s, SemCache creates a new entry for the range. If the end address of the found entry is greater than or equal to e, it returns the entry. If the end address of the found entry is greater than s and less than e, SemCache iterates through the subsequent entries until it finds all the entries that overlap with the current range. It then merges the ranges together, performing appropriate data movement operations so that the eventual state of the new entry is C.

4.2.1 Managing available GPU memory

The amount of data sent to the GPU might be too large to fit the available GPU memory. In such a situation, to allocate new data in the GPU memory, cached data must to be freed. To determine which address ranges should be freed, SemCache uses least-recently-used (LRU) policy. Any data accessed on the GPU is added to the end of a queue. If the GPU memory is full, data at the head of the queue is removed. Note that depending on the application other polices can be used. Multiple replacement polices can be easily integrated with SemCache and the programmer can have the option to choose between them.

4.2.2 Determining Granularity

SemCache by itself is simply a variable-granularity cache that supports a few methods to transfer data between the CPU and GPU. The key to SemCache's utility is that the granularity of the cache is determined by the semantics of the GPU libraries being used in a program. In particular, we note that the address ranges tracked by the cache are determined during cache lookup: if a particular range has not been seen before, a new entry for that range is created. Hence, if a library call takes as input matrices `A` and `B` and produces as output a matrix `C`, the three matrices can be individually tracked by performing lookups on their address ranges. For example, if `A` were an $n \times n$ matrix (of floats), invoking `lookup(A, A + sizeof(float) * n * n)` would cause SemCache to start tracking matrix A, and whether it existed on

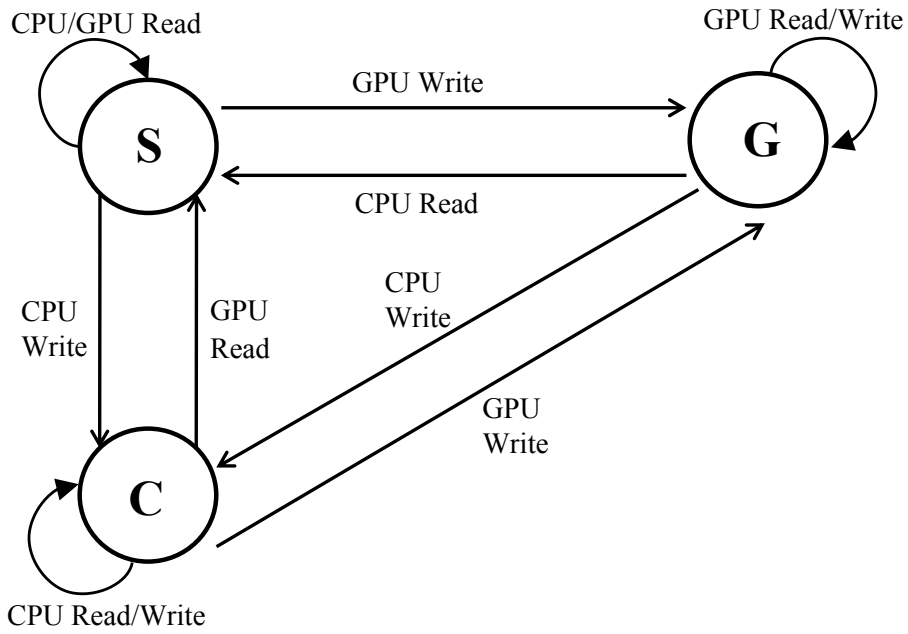

Fig. 4.2. Write-back protocol (States: GPU/CPU/Shared)

the GPU or not. Note that the current implementation of SemCache only tracks contiguous memory ranges; data structures that are not contiguous ranges have to be tracked with multiple entries.

4.3 SemCache Instrumentation and Protocols

4.3.1 Write-back Protocol

The interfaces of SemCache can be used to implement a basic protocol to manage data movement between the CPU and GPU. The protocol tracks reads and writes on both devices, and transfers data when necessary. Figure 4.2 shows the basic protocol, which behaves similarly to an MSI coherence protocol. Data that is computed on the GPU remains on the GPU until the CPU needs to read it. Similarly, data computed on the CPU remains on the CPU until the GPU needs it. If either the CPU or GPU

writes a piece of data, that data is invalidated on the other device. Adopting the terminology of Quintana-Orti *et al.*, we call this a *write-back* protocol [7].

To implement this protocol, SemCache provides an API that can be called before CPU or GPU reads and writes to a range of data. The API looks up entries in the caching directory and performs communication if required based on the data status.

writeCPU(s, e) Execute before writing CPU address range [s,e). It Looks up and retrieves the translation record associated with memory range $[s, e)$. If the status is G, it is transferred back to the CPU. Finally, it is invalidated and the status is set to C.

readCPU(s, e) Execute before reading CPU address range [s,e). It Looks up and retrieves the translation record associated with memory range $[s, e)$. If the status is G, it is transferred back to the CPU. The status is set to S since it is read only.

writeGPU(s, e) Called after a GPU method that writes [s,e). It Looks up and retrieves the translation record associated with memory range $[s, e)$. The status is set to GPU only (G).

readGPU(s, e) Called before reading a GPU address range [s,e). It Looks up and retrieves the translation record associated with memory range $[s, e)$. If the status is G, it is transferred back to the CPU. The status is set to shared (S).

Although `writeGPU` and `readGPU` can be easily embedded inside the GPU library calls, `writeCPU` and `readCPU` are harder to insert before CPU reads and writes without modifying the original code. SemCache runtime system can automatically invoke the CPU reads and writes without modifying the original code as discussed in Section 4.3.4. Additionally, Section 5.5.2 discusses how a library writer can use these interface methods to manage data movement for a particular library.

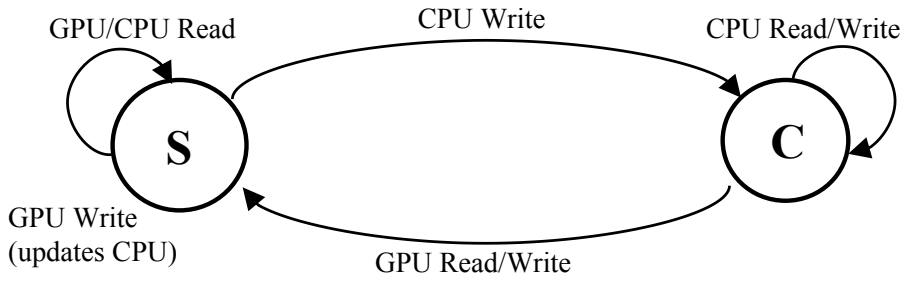

Fig. 4.3. Write-through protocol (States: CPU/Shared)

4.3.2 Write-through Protocol

We introduce a further protocol simplification. Because reads on the CPU are much more prevalent than writes, and because most results computed by the GPU are eventually needed on the CPU, we eagerly transfer any data *written* by the GPU during a library operation back to the CPU. This affects how library operations that modify data are handled. In the write-back protocol, `writeGPU` is invoked to invalidate the data on the CPU. In the write-through protocol, `writeGPU` is never invoked, but instead `readCPU` is immediately called to transfer the data back to the CPU. Section 5.5.2 gives a concrete example of how the implementation of a library changes based on the protocol.

In the write-through protocol, data is never in the G state; it can only be in C or S. The simplified protocol is shown in Figure 4.3. Because data is eagerly written back to the CPU, we again adopt previous terminology and call this a *write-through* protocol [7]. Note that because data is never in the G state, we no longer need to instrument CPU reads, reducing instrumentation overheads.

4.3.3 SemCache in Practice

Figure 4.4 shows the data movement performed by our system on the simple example of Figure 1.1 using the two different protocols. We note that when using

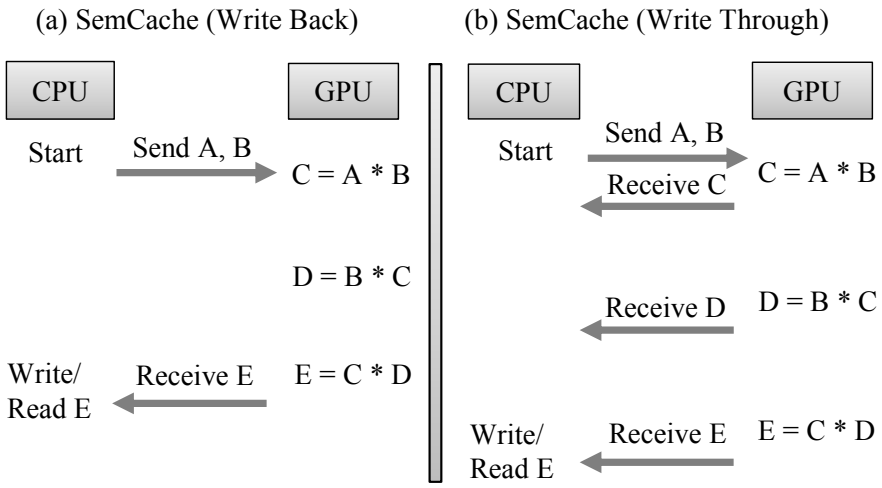

Fig. 4.4. SemCache communication model

the write-back protocol (Figure 4.4(a)), SemCache performs the minimum required data movement (*cf.* Figure 2.5(b)). At the first invocation of matrix-multiply, `A` and `B` are transferred to the GPU, and SemCache tracks them in S state. When `C` is computed, it is tracked in G state. Because both `B` and `C` are current on the GPU, later invocations of matrix multiply need not perform any more data transfer. Finally, `E` will be transferred back to the CPU once that matrix is read by other portions of the program. In the write-through protocol (Figure 4.4(b)), the amount of communication from the CPU to the GPU is minimal. However, because GPU results are eagerly communicated back to the CPU, we see that some extra communication is performed from the GPU to the CPU.

Crucially, because all of the necessary instrumentation is either automatically inserted or encapsulated in a GPU library (see Section 5.5.2), programmers can simply use SemCache-enhanced GPU libraries as drop-in replacements for their existing libraries.

4.3.4 Instrumenting CPU Reads and Writes

SemCache provides two approaches to inserting instrumentation to implement the write-back and write-through protocols: statically-inserted instrumentation (either by the programmer or the compiler), and dynamic instrumentation using the operating system's page-protection facilities.

Statically-inserted Instrumentation

Conservatively, the programmer or compiler must guard every read or write on the CPU with appropriate instrumentation. In practice, because data movement between the CPU and GPU can only occur when GPU libraries are invoked, simple analyses can reduce this instrumentation overhead. For example, any data that will never be sent to the GPU (*i.e.*, can never be passed to a method call executed on the GPU) does not need to be instrumented. Furthermore, reads or writes to array locations that occur in loops can be guarded by a single call, with the parameters determined by array analyses that determine what portions of an array are accessed in a loop. These analyses, of which many exist, are beyond the scope of this thesis; we assume that such an analysis has already been performed, allowing array accesses to be efficiently guarded.

The run-time nature of SemCache tolerates instrumentation imprecision. In particular, looking up address ranges that are not shared with the GPU does not introduce extra communication, nor does invalidating the same range more than once; these operations merely introduce extra caching overhead. Conservatively invalidating an address region is also safe: while this unnecessary invalidation causes unnecessary communication, it does not affect the correctness of the program.

Note also that although we instrument particular reads and writes, as well as particular GPU operations, to perform our caching, the cache lookups, etc., are based on address ranges. As a result, program behaviors such as aliasing do not present

correctness problems; the caching is performed based on the underlying memory, not the specific name given to that memory.

Even after removing unnecessary instrumentation through the above analyses, and avoiding the instrumentation of reads on the CPU with the write-through protocol, invoking `writeCPU` before every write to data that may reside on the GPU still introduces unnecessary instrumentation. For example, on a write to data that has already been invalidated on the GPU, it is redundant to look up the data and "re-invalidate" it. Developing an analysis to remove redundant instrumentation is a subject for future work.

Page-protection-based Instrumentation

Rather than using statically-inserted instrumentation of CPU reads and writes, SemCache can also use the operating system's virtual memory facilities to implement the write-back and write-through protocols. The OS Memory protection is a typically used to control memory access rights on a computer. Since the memory is organized as pages, each page can be hold one of three states: no access, read only and read/write. If a process accesses a protected page, the system triggers a page fault.

In SemCache, the page protection mechanism can be used to automatically invoke `readCPU` and `writeCPU`. For each data structure that SemCache tracks on the CPU, SemCache sets page protection flags for all the pages the data structure spans. The page protection flags are set according to the state of the data structure as follows:

- If the structure is in G state, its pages are set to `PROT_NONE`;
- if the structure is in S state, its pages are set to `PROT_READ`;
- if the structure is in C state, the pages are set to `PROT_READ` | `PROT_WRITE`.

If a CPU access triggers a page fault, SemCache invokes `readCPU` or `writeCPU` based on the required operation and the current state as follows:

- Write to a no access region or a read only region; it triggers `writeCPU` which looks up the address that caused the page fault in the caching directory. If the translation record is found, it transfers the matrix back from the GPU and the submatrix status becomes CPU only (C). The structure's page protection flags are set to read/write.

- Read from a no access region; SemCache invokes `readCPU` which transfers the submatrix back from the GPU and the submatrix status becomes shared (S). The structure's page protection flags are set to read only.

Note that although *detection* of accesses that require communication occurs at the page granularity, *communication* does not: if a structure needs to be communicated from the GPU to the CPU, SemCache transfers the entire structure, and changes the status of all of the associated pages on the CPU. This preserves SemCache's variable-granularity advantages over systems like CUDA unified memory.

The main advantage of page-based invalidations over statically-inserted invalidations is that these invalidations are handled fully automatically; no additional instrumentation or compiler analysis is required. However, they also have some disadvantages as well: to work correctly with page-protection operations, and to avoid false sharing issues, page-based invalidations require that a program's memory layout must be changed to ensure that all data structures that may be communicated to the GPU are page-aligned and padded out to page boundaries.

We note that this page-based strategy is similar to that used by DyManD [22]. However, unlike DyManD, SemCache still tracks data structures and maintains mappings between the CPU and GPU according to semantic information, rather than requiring direct memory mapping between the CPU and GPU. In addition to allowing programs whose working sets exceed GPU memory, SemCache's approach allows for *semantic links* to be formed between data on the CPU and data on the GPU, as the next section explains.

4.4 Semantic Mapping with SemCache

This section discusses how the basic principles of SemCache can be extended and generalized to achieve additional savings. In particular, we describe how ancillary structures can be added to SemCache to allow it to "cache" the results of arbitrary functional computations, essentially allowing SemCache to serve as a memoizing service for GPU computations. This facility can be used for many purposes, from avoiding expensive recomputations (*e.g.*, storing only the factorized forms of matrices on the GPU) to performing data layout transformations (*e.g.*, mapping row-major data structures on the CPU to column-major layouts on the GPU). In essence, instead of directly mapping between CPU and GPU data, SemCache can create a *semantic link* between data on the CPU and a transformed version of that data on the GPU.

To see how SemCache can create these semantic links, we note that memoization effectively maps a particular input of a function to its pre-computed output. That is, for a function $f : X \to Y$, a memoized input x is used to look up its previously-computed output y, rather than evaluating $f(x)$. If we consider x as data residing on the CPU, and y as data residing on the GPU, then we can abstract SemCache's default behavior as simply the memoization of the identity function $f(x) = x$. For a given input (*i.e.*, data on the CPU), SemCache provides the previously-computed (*i.e.*, previously-communicated) output (*i.e.*. data on the GPU). In other words, SemCache is indexed by inputs on the CPU and provides a map to the results of the identity function stored on the GPU.

We can see that there is no need for SemCache's operation to be restricted to memoizing the identity function on to the GPU. The results of other functions can be memoized as well. Consider performing matrix factorization (*e.g.*, LU factorization) as an intermediate step in equation solve (GESV), the factorization is not saved. Such factorizations on the GPU are time consuming, so repeatedly factorizing a matrix can be wasteful. Instead, we can use an extended version of SemCache to cache the

results of the factorization on the GPU, instead of just the inputs to the factorization operation.

Figure 4.5 shows how SemCache is extended. The same address ranges tracked by the baseline cache are used to index into a computation cache, which stores the GPU location of the *results* of a particular computation. Since this data is computation-specific, each type of computation to be memoized will need separate lookup tables. Note that the computation structures need not separately track the status of the data. If the data in the main cache is ever invalidated on the GPU (*i.e.*, its status is changed to C), the corresponding entries in any computation caches are simply removed.

To attempt to skip performing a GPU computation on an address range $[s, e)$, SemCache takes the following steps. First, the range is found in the main cache. If the status of the range is S or G, the lookup is repeated in the computation cache, and, if a result is found, the GPU computation can be elided. If the status of the range is C, or there is no entry in the computation cache, the GPU computation is performed, the status of the range is set to S, and the computation cache is updated.

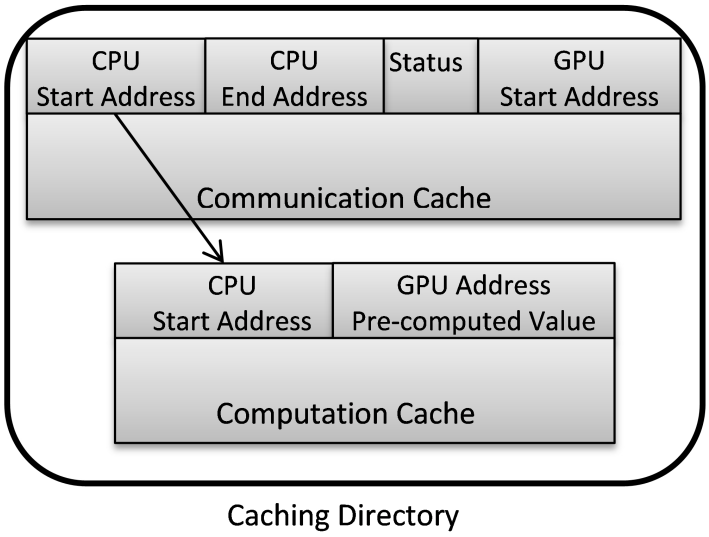

Fig. 4.5. Caching Directory Components

We note that the particular set of lookups, and particular data stored, is based on the semantics of the computation being cached. Using SemCache to create semantic links hence requires adding instrumentation to GPU libraries to perform the necessary lookups, etc. Nevertheless, this instrumentation can be completely encapsulated in a library, and its effects need not be visible to the programmer, preserving the library as a drop-in replacement.

4.5 Implementation

To demonstrate how SemCache can be used to improve the performance of GPU computation libraries, we use it to produce a drop-in replacement for BLAS. This allows programmers to replace BLAS calls in their code with calls to our library, automatically offloading computation to the GPU and handling memory management transparently. The GPU kernels of our library are based on the corresponding CUBLAS implementations. Our library supports either the write-back protocol or the write-through protocol, controlled by a compile-time flag. The implementation is done in C on a LINUX OS. Since page-based invalidations are used, `malloc` calls must be modified to page-aligned allocations and padded to page boundaries to avoid false sharing, as described in Section 5.3.2. To provide page-aligned allocations, `valloc` can be used instead of `malloc` and the size of data can be padded to page boundaries using address masking.

Figure 4.6 shows the sequence of calls that would be required to use CUBLAS to perform matrix multiply, with all communication explicitly managed by the programmer. In contrast, Figure 4.7 shows the interface for the SemCache-enhanced version of matrix multiply.

Figure 4.8 shows SemCache API to implement the write back protocol. The `writeGPU` and `readGPU` API are embedded inside the GPU library calls as shown in Figure 5.6. The API looks up entries in the caching directory and performs commu-

```
1  cudaMalloc(A) //Allocate space on device mem.
2  cudaMalloc(B) //Allocate space on device mem.
3  cudaMalloc(C) //Allocate space on device mem.
4
5  cublasSetMatrix(A) //Move matrix A to device
6  cublasSetMatrix(B) //Move matrix B to device
7  cublasSetMatrix(C) //Move matrix C to device
8
9  cublasDgemm(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
10
11 cublasGetMatrix(C) //Get matrix C from device
```

Fig. 4.6. Matrix multiply using CUBLAS code

```
1  SemCacheDgemm(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC
       )
```

Fig. 4.7. SemCache library interface

```
1  //execute before writing CPU address range [s,e)
2  TranslationRecord writeCPU(s, e) {
3    entry = lookup(s, e);
4    if (entry.status == G) //CPU data not current
5      transferToCPU(entry);
6    invalidateOnGPU(entry);
7    return entry;
8  }
9
10 //execute before reading CPU address range [s,e)
11 TranslationRecord readCPU(s, e) {
12   entry = lookup(s, e);
13   if (entry.status == G) //CPU data not current
14     transferToCPU(entry);
15   mprotect(s, entry.size,  PROT_READ);
16   return entry;
17 }
18
19 //called after a GPU method that writes [s, e)
20 TranslationRecord writeGPU(s, e) {
21   entry = lookup(s, e);
22   invalidateOnCPU(entry);
23   mprotect(s, entry.size,  PROT_NONE);
24   return entry;
25 }
26
27 //called before a GPU method that reads [s, e)
28 TranslationRecord readGPU(s, e) {
29   entry = lookup(s, e);
30   if (entry.status == C){ //GPU data not current
31     transferToGPU(entry);
32     mprotect(s, entry.size,  PROT_READ);
33   }
34   return entry;
35 }
```

Fig. 4.8. Operations to implement write-back protocol

nication if required based on the data status. Any data replicated on both devices is protected on the CPU to prevent future access to it. Any future access will have to go through the coherence protocol to make sure the data on the device is up to date. The POSIX operating system provides a method called `mprotect` which changes the protection on region of memory (spanned by multiple pages) The protection can be set to no access, read only and read/write. The API `writeCPU` and `readCPU` are automatically invoked when a page fault occurs using SemCache runtime system as shown in 4.10. Note that these methods return the translation record, as invoking methods on the GPU may require knowing the addresses where the necessary data is stored.

Figure 5.6 shows how matrix multiply is implemented. Under the hood, we still invoke the CUBLAS matrix multiply method. However, all communication is managed by SemCache, and is only performed when necessary. When SemCache is called, the caching directory is searched for each matrix using the start and end address in the main memory. The start address is the pointer address and the end address is calculated using the matrix size. The cache keeps a record of the start and end address in the main memory for each matrix accessed using our library. If the matrix does not exist, it is transferred to the GPU and cached. A new record is created for it in the cache. If the matrix is found in the cache and it is in S state, then it is a hit and there is no need to transfer the matrix to the GPU. The matrix address in the GPU memory is taken from the translation record. This address is used to access the matrix using CUBLAS. If the matrix is not valid on the GPU (it is in C state), it is transferred to the GPU and the record in the cache is updated. After all of the matrices are transferred or located in the GPU memory, the CUBLAS call is executed. Then the result is transferred back to the main memory.

CUBLAS does not provide an implementation of general equation solve (GESV), instead only providing triangular solves for factorized matrices. While there exist several efficient GPU implementations of LU factorization [4,32], our implementation instead implements equation solve in two steps: we compute the LU factorization on

```
SemCacheDgemm(TRANSA,TRANSB,M,N,K,ALPHA,
    A,LDA,B,LDB,BETA,C,LDC)
{
  //A stored on CPU in memory range [A, A+(M*K*8))
  //A will be read by GPU, its state will be "S"
  entryA = readGPU(A, A + (M*K*sizeof(double)));

  //B stored on CPU in memory range [B, B+(K*N*8))
  //B will be read by GPU, its state will be "S"
  entryB = readGPU(B, B + (K*N*sizeof(double)))

  //C stored on CPU in memory range [C, C+(M*N*8))
  //C will be read by GPU, its state will be "S"
  entryC = readGPU(C, C + (M*N*sizeof(double)))

  cublasDgemm(TRANSA,TRANSB,M,N,K,ALPHA,
     entryA.gpu_s,LDA,
     entryB.gpu_s,LDB,BETA,
     entryC.gpu_s,LDC)

  //C was written by cublasDgemm
#ifdef WRITEBACK
  //If we're using write-back, writeGPU must be called to
     invalidate, C state will be "G"
  writeGPU(C, C + (M*N*sizeof(double)))
#else
  //If we're using write-through, we eagerly communicate to
     the CPU, C state will be "S"
  readCPU(C, C + (M*N*sizeof(double)))
#endif
}
```

Fig. 4.9. Implementation of SemCache matrix multiply (DGEMM)

the CPU, then perform the equation solve on the GPU using CUBLAS's triangular-solve routines. We then use SemCache's computation caching capability to avoid performing the factorization whenever possible. This implementation was chosen to demonstrate SemCache's generalized memoization ability.

Figure 4.10 shows how the page fault handler is initialized and used. The `Init` function binds the page fault with the handling function. It is called once at the initialization of the program. At runtime, when a page fault occurs `fault_handler` function is executed. The function has the address where the page fault happened. A lookup in SemCache caching directory is performed using this address. If the address falls into the range of any of the tracked data structures, the translation record for that data structure is returned. The protection is removed from the data to prepare it for receiving the updated data. Then, `readCPU` or `writeCPU` are invoked based on the page fault type. The page fault type can be determined by inspecting a special register. The register has three values each has a special meaning:

Value=4 The memory is set to no Access, the CPU needs read access.

Value=6 The memory is set to no Access, the CPU needs write access.

Value=7 The memory is set to read only, the CPU needs write access.

There is an overloaded version of `readCPU` or `writeCPU` which takes the translation record as an input instead of looking up the data again.

Using SemCache with complex memory structures

SemCache current implementation supports contiguous data structures. However, SemCache low level API (readGPU and writeGPU) can be used to offload non-contiguous data structures (i.e., trees and graphs) by invoking the appropriate methods on each address range for the data structure. SemCache will automatically transfer and track data. It becomes the library writer's responsibility to build a drop-in library using SemCache API.

```
1  static void fault_handler(int sig, siginfo_t *si, void *uap)
2  {
3    ucontext_t *context =  (ucontext_t *) uap;
4    //Get the type of the page fault from the registers
5    int page_fault = context->uc_mcontext.gregs[REG_ERR];
6    //lookup the page fault address in the caching directory and
         return the corresponding translation record if found
7    TranslationRecord translationRecord =
8    lookupInCacheDir(si->si_addr);
9    //remove the page protection for the entire data size
10   mprotect(translationRecord.HostStartAddress,
        translationRecord.Size_Padded,  PROT_READ|PROT_WRITE)
11
12   if (page_fault == 4) { //No Access -> Needs Read Access
13     readCPU(&translationRecord);
14   }
15   else if (page_fault == 6) { //No Access -> Needs Write
        Access
16     writeCPU(&translationRecord);
17   }
18   else if (page_fault == 7) {//Read Only -> Needs Write Access
19     invalidateOnGPU(&translationRecord);
20   }
21 }
22
23 static void Init() //InitHandler
24 {
25   struct sigaction sa;
26   sa.sa_flags = SA_SIGINFO;
27   sigemptyset(&sa.sa_mask);
28   //define the fault handler function
29   sa.sa_sigaction = fault_handler;
30   //bind the fault handler
31   if (sigaction(SIGSEGV, &sa, NULL) == -1){
32     perror("sigaction");
33     exit(EXIT_FAILURE);
34   }
35 }
```

Fig. 4.10. Implementation of SemCache Page Fault Handler

4.6 SemCache Experimental Evaluation

Experiments were run on a server with 24 AMD Opteron 6164 HE Processors (1.7 GHz, 512 KB L2 cache), 32 GB memory, running 64-bit Fedora Linux, and NVIDIA Tesla C2070 card (6 GB memory) with a peak memory bandwidth of 144 GB/s. Three libraries were used: CUBLAS version 4.0, CULA Dense R15 and MAGMA version 1.2. Each test was run 3 times, distributed over a wide range of time, on an unloaded machine and the median time selected.

We evaluated our library in two ways. First, we used a test case based on a series of matrix multiplications (as in Figure 1.1). The simplicity of the test case allowed us to perform several comparisons with other libraries, as well as test the two SemCache protocols. Nevertheless, the primary target for our work is large-scale computational applications where hand-tuning is infeasible. To study SemCache's effectiveness in this setting, we used our modified BLAS libraries (see Section 5.5.2) on a large-scale, real-world computational mechanics application, which uses finite element methods and domain decomposition to solve a structural dynamics problem.

As described in Section 4.3, SemCache can perform invalidations either with statically-inserted instrumentation at CPU reads and writes, or using a page-protection-based mechanism. We found empirically that the two approaches perform equivalently; in the experiments presented here, we use page-protection automatic instrumentation to perform invalidations.

4.6.1 Matrix Multiplication Test Case

Figure 5.8 shows the total execution time for the test case. The results are collected using CUBLAS, CULA Standard Interface (which automatically manages communication between the CPU and GPU), MAGMA and SemCache using both write-back and write-through policies. Communication in CUBLAS and MAGMA are hand tuned. The total execution time is normalized to CUBLAS execution time. The best performance is achieved by hand tuning the memory transfers using CUBLAS.

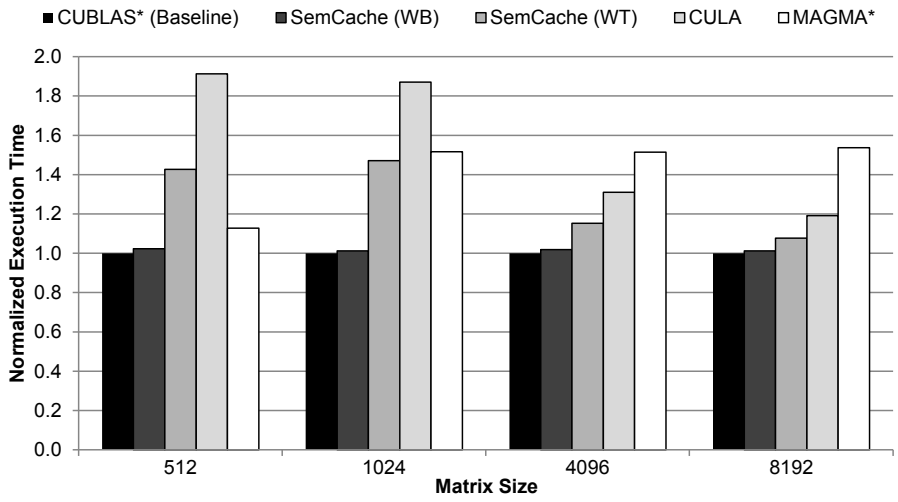

Fig. 4.11. Test case normalized execution time *(Communication in CUBLAS and MAGMA is hand optimized)

CULA performance was slowed down due to the repeated unnecessary transfers. SemCache write-back performance matches the optimal communication performance using CUBLAS, but is slightly slower due to caching overhead. SemCache write-through performance is the next closest to the optimal communication performance. The slowdown is due to the eager copying back of the result to the CPU after each multiplication. MAGMA's performance varies based on the matrix size (as it uses different kernels tuned to different matrix sizes), but overall uses slower implementations than CUBLAS.

Communication savings Figure 5.9 breaks down the communication performed for a medium-sized squared matrix (N=4096), distinguishing between data sent and data received. We collected data for CUBLAS, MAGMA, SemCache write-back, SemCache write-through and CULA. Hand-tuned communication for CUBLAS and MAGMA minimize the memory transfers. SemCache write-back performs exactly as much communication as hand-tuned libraries. It performs the minimum amount of data transfers, as the data is already cached on the GPU and is only sent back when needed. In SemCache write-through, the data sent to the GPU is minimized.

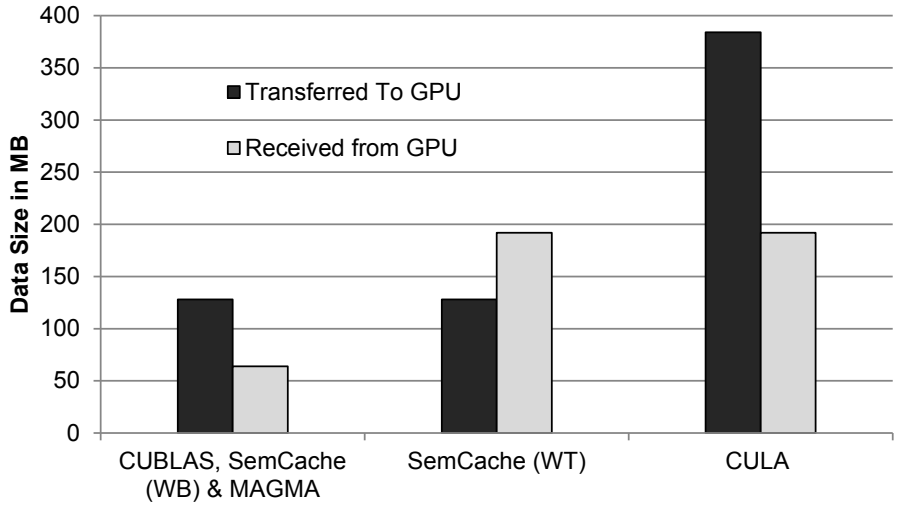

Fig. 4.12. Test case communication results (N=4096)

However, data is always copied back, introducing redundant communication. CULA shows the most overhead since matrices are sent to the GPU for every calculation. The results are also sent back to main memory after each calculation, introducing extra communication.

4.6.2 Computational Mechanics Case Study

We next tested SemCache's performance in a real-world setting. We studied a large computational mechanics application [33]. In this application, domain decomposition is used for the simulation of structural dynamics problems. Domain decomposition methods solve a boundary value problem by splitting it into smaller boundary value problems on subdomains and iterating to coordinate the solution between adjacent subdomains. Then the Newmark-beta method of numerical integration is used to solve differential equations. The application we use solves the subdomains recursively. This method was introduced by [34]. Typical structural dynamics problem include simulation of the effect of cracks in structures, or buildings under stress.

Most of the application's execution is spent performing linear algebra routines. Three main double-precision linear algebra subroutines are used: matrix multiplica-

tions (DGEMM) and scalar multiplication/vector addition (DXPY) to couple and update the subdomains results and equation solve (DGESV) to solve the system of equations at each node. Because these operations make up a large fraction of the application's computation, they are attractive targets for offloading. However, optimizing communication in this application is essentially impossible. The application has tens of thousands lines of code, and the relationship between various linear algebra operations is difficult to reason about due to recursive calls and multiple abstraction layers.

We evaluated five versions of this application. A serial CPU version that performed no offloading, a CUBLAS version with hand-inserted unoptimized communication (communication can't be optimized manually due to program abstraction), a CULA version that simply replaces all CPU BLAS calls with CULA BLAS calls, a version using our SemCache write-through library, and another version using our SemCache write-back library.

The SemCache versions of the application exploit computation caching in two ways. First, as described in Section 5.5.2, our implementation of equation solve leverages SemCache's computation-saving capabilities to memoize the results of matrix factorization. Second, the baseline CPU version of the application uses row-major storage for its matrices, while CUBLAS assumes column-major storage. SemCache thus creates a semantic link between the two representations, avoiding performing the transformation unless the data changes[2].

We used three inputs with different characteristics, ranging across various sizes: Rocket32, which has 7262 nodes and takes 246 seconds to run on the CPU; Cube14, which has 3375 nodes and takes 130 seconds to run on the CPU; and Cube10, which has 1331 nodes and takes 10 seconds to run on the CPU.

[2]Because the row-major/column-major transformation is only necessary due to an implementation detail of the original application, we factor out the transformation time for the non-SemCache versions in all our results.

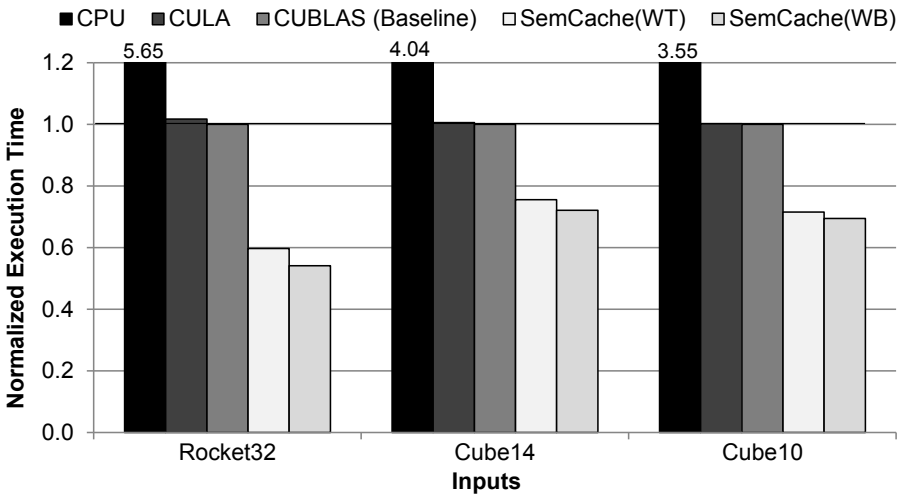

Fig. 4.13. Testing application normalized execution time

Execution time

Figure 4.13 shows the total execution time for the five variants of the application, across the three inputs. Run time is normalized to the CUBLAS variant. All inputs gained from three to six times speedups when running on the GPU over the CPU version. CULA and CUBLAS performance was very similar. CULA uses CUBLAS as an underlying library with a few additional optimizations. Both approaches incur the cost of extra communication. Using SemCache with write-through policy, the performance improved (30% to 40%) over the GPU CUBLAS baseline version due to the communication savings from caching. SemCache with write-back gained an additional 4–10% over write-through, as data was only transferred back to the CPU when needed. The inputs speedup ranges are different based on the structure of input and domain decomposition. Inputs whose subdomains have larger shared interfaces generate more matrices that will be repeatedly reused. As a result, caching yields more benefits.

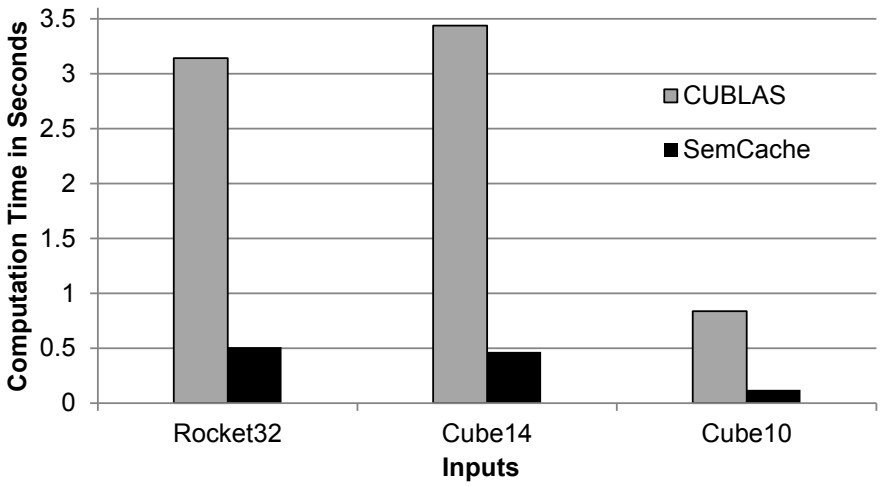

Fig. 4.14. Computation time for factorization

Communication savings

Table 4.1 shows the amount of data transferred to the GPU. The SemCache results show the optimal communication for the application since all of the calculations were computed on the GPU and the hit rate was 100%. Using SemCache, more than 80% of the unoptimized communication is reduced. Both write-through and write-back policies reduced the size of the data sent to the GPU. Write-back policy reduced the size of the data received from the GPU. These savings are a result of eliminating redundant transfers since the data in the testing application is shared between different subdomains. The same matrices will be reused multiple times for different subdomains.

Computation caching

We evaluated the savings of performing computation saving for LU factorization. Figure 4.14 shows the LU factorization time on the CPU for out testing application. Using SemCache, repeated computations are eliminated since the factorized

Table 4.1
Size of transferred data using CUBLAS versus SemCache (in GB)

Input/Library	CUBLAS		SemCache	
	Sent	Received	Sent	Received
Rocket32	23.70	5.58	2.02	2.45
Cube14	10.67	1.53	1.01	0.63
Cube10	3.01	0.33	0.29	0.13

Table 4.2
Data transfer time from CPU to GPU for CUBLAS versus SemCache with overhead (in seconds)

Input/Library	CUBLAS	SemCache	
	Transfer	Transfer	Caching Over.
Rocket32	11.09	0.86	0.38
Cube14	5.27	0.47	0.05
Cube10	1.32	0.12	0.023

Table 4.3
Operations count at runtime

Input/Op.	GEMM	GESV	XPY	COPY	Lookup
Rocket32	6720	1209	3520	480	30578
Cube14	944	233	688	104	4882
Cube10	470	131	394	62	2584

matrices are already cached. Fewer factorizations are needed, which reduces the total computation time by more than 80% .

Caching overhead

Table 4.2 shows the data transfer time to GPU for different inputs. The results show that the caching overhead is very low (less than 4% of SemCache total runtime). The overhead comes mainly from searching and updating the cache directory. The transfer time using our library including the caching overhead is significantly less than the transfer time for CUBLAS without caching. We note, however, that our low caching overhead is due to SemCache's variable granularity, which requires fewer invalidations and fewer lookups.

Instrumentation statistics

For our testing application, more than 10,000 lines of code and around 45 BLAS and LAPACK calls are used. No writeCPU invalidations were needed because the all of the calculations were computed on the GPU. For the write back protocol, reads were instrumented using readCPU API. Seven API calls were needed.

Table 4.3 shows how many times matrix multiply (GEMM), equation solve (GESV), scalar multiplication and vector addition (XPY), copy (COPY), lookup and invalidation operations were executed. The lookup matches exactly the number of matrices sent to the GPU (3 per GEMM, 2 per GESV, XPY and COPY).

Applicability

As discussed in the implementation section, SemCache is used as a drop-in library. It simplifies programing GPUs and saves the programmer time. For example, offloading each BLAS call to the GPU requires at least 5 lines of code (3 lines transferring the matrices to the GPU, 1 line performing the computation using CUBLAS and 1 line sending the data back). This operation can be done in a single line using SemCache. The testing application, has 45 BLAS and LAPACK calls. Offloading these calls to GPUs using CUDA requires at least 225 lines of code where doing this using

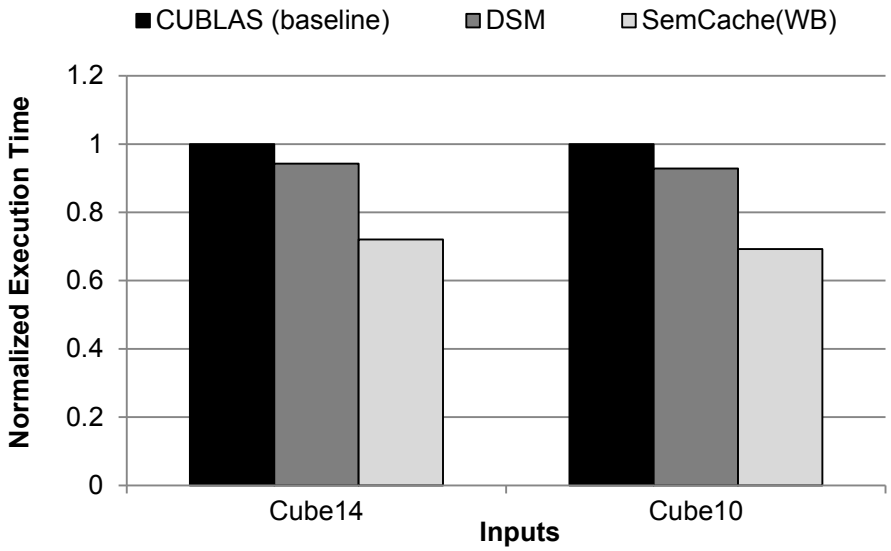

Fig. 4.15. Testing application normalized execution time for CUBLAS, SemCache write-back and DSM

SemCache requires no code changes since the CPU calls can be dynamically linked with SemCache library.

Comparison with fixed-granularity approaches

One of the primary advantages of SemCache over distributed-shared memory systems is its ability to track data and perform communication with variable granularity, rather than using a fixed granularity. To quantify this benefit, we modified the page-protection version of SemCache (Section 5.3.2) to perform communication in page-sized chunks, rather than tracking entire data structures[3]. Figure 4.15 compares the CUBLAS baseline with this DSM-like approach as well as SemCache's variable-granularity approach on our case study[4]. Clearly, fixed-granularity tracking does not perform as well as SemCache.

[3]This variant is not strictly correct, as without transferring data at matrix granularity, the semantic mapping between row-major and column-major representations cannot be maintained. Nevertheless, this variant lets us explore the penalty of page-granularity caching.

[4]Due to limitations of the page-based approach, large inputs (such as Rocket32) cannot be run.

Interestingly, the total amount of data communicated is the same for both the fixed-granularity and variable-granularity versions. The performance difference arises because fixed-granularity tracking breaks that communication into more discrete communication operations, incurring additional overhead. Clearly, taking advantage of semantic information to perform variable-granularity tracking and communication yields a notable performance benefit.

4.6.3 Linpack Benchmark

We also tested SemCache's performance with High-Performance Linpack Benchmark for Distributed-Memory Computers which is often used to benchmark supercomputers. Linpack benchmark is a software package that solves a (random) dense linear system in double precision arithmetic on distributed-memory computers. The application uses BLAS library for matrix multiplication (DGEMM) and equation solve (DTRSM) computations.

Since SemCache currently supports one GPU, we ran the application using a single process. Different problem sizes (N) were used with varying block size (NB). The results are collected using SemCache write-back policy.

Execution time

Figure 4.16 shows the total execution time for Linpack using different problem sizes. All inputs gained from 13 to 16 times speedups when running on the GPU using CUBLAS over the CPU version. Using SemCache the performance improved (3% to 8%) over CUBLAS due to the communication savings from caching.

Communication savings

Figure 4.17 shows the amount of data transferred to the GPU. Using CUBLAS to offload individual kernels results in redundant transfers. Using SemCache, more

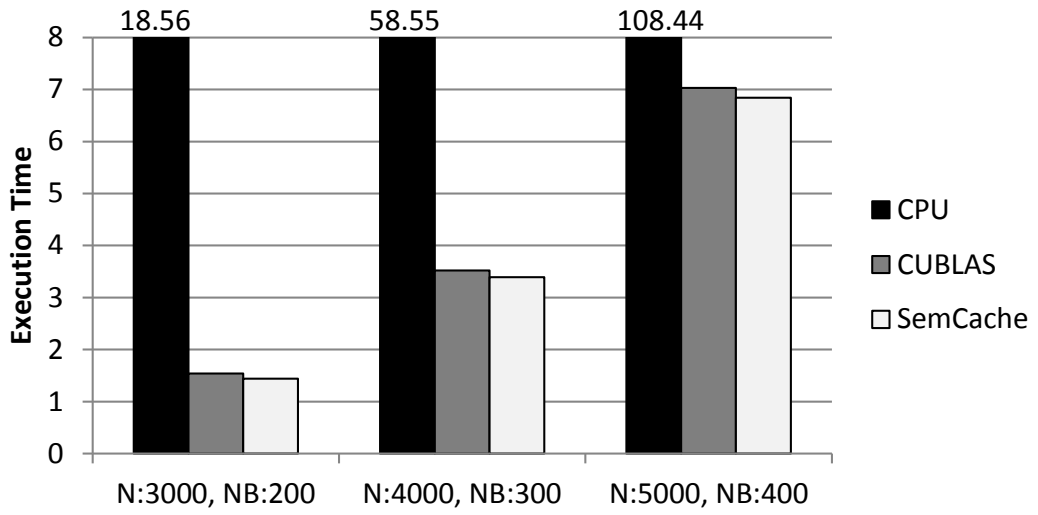

Fig. 4.16. Linpack execution time

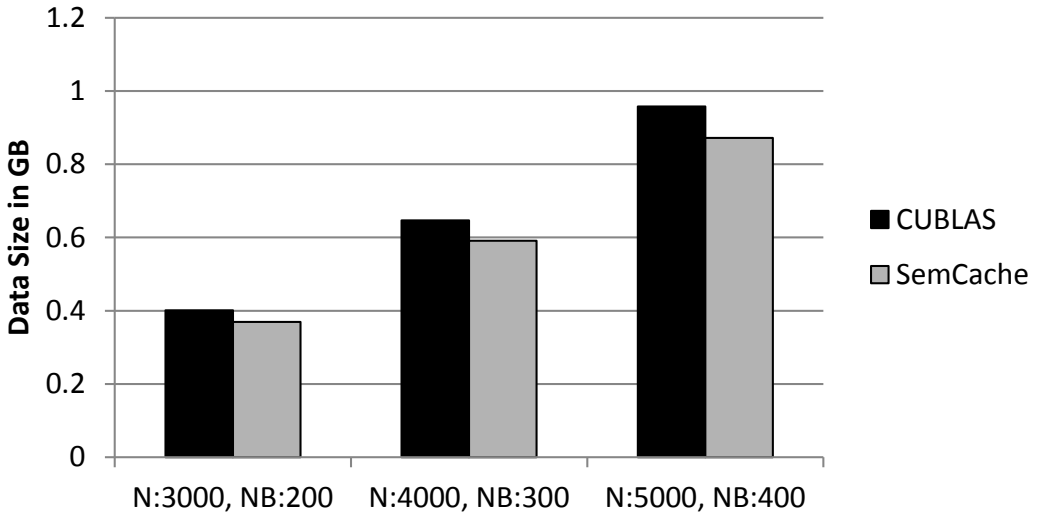

Fig. 4.17. Size of transferred data to GPU using CUBLAS versus SemCache (in GB)

than 7% to 8% of the unoptimized communication is reduced. Savings were achieved from passing the result of the equation solve (DTRSM) subroutine to the matrix multiplication (DGEMM) subroutine.

5. SEMCACHE++

This section introduces SemCache++ [3], an extension of SemCache (described in Chapter 4) that supports multiple GPUs. SemCache++ automatically manages data movement and synchronization across SemCache++-enabled library calls. These libraries can thus be used as direct replacements for CPU libraries, providing the performance of hand-tuned multi-GPU implementations without breaking the abstraction boundaries of the library.

Unlike a single GPU implementation, there are many challenges in multi-GPU implementation. The first challenge is workload distribution and scheduling between multiple GPUs. There are multiple parallel algorithms for solving matrix computations on distributed systems. Choosing the right algorithm depends on the underlying network and computing architecture. Many factors can be taken into consideration to determine which distribution and scheduling algorithm should be used such as: load balancing, optimizing communication and computation-communication ratio. In addition to that, communication between different devices needs to be managed: CPU to multi-GPU communication, GPU to GPU communication, multi-GPU to CPU communication. All of these challenges make it very hard to program multiple GPUs. The code to offload a single kernel to the GPU will become a program by itself if expanded to multiple GPUs which makes it clear that abstraction is needed and all complexities should be managed automatically inside the runtime system.

SemCache requires many changes to support multi-GPU offloading. Firstly, a decomposition algorithm is used to divide the problem into subproblems. The algorithm is chosen to maximize the communication computation overlapping. Secondly, the caching directory will be modified to track submatrices on multiple GPUs and their status. Thirdly, asynchronous transfers are scheduled using streams to allow

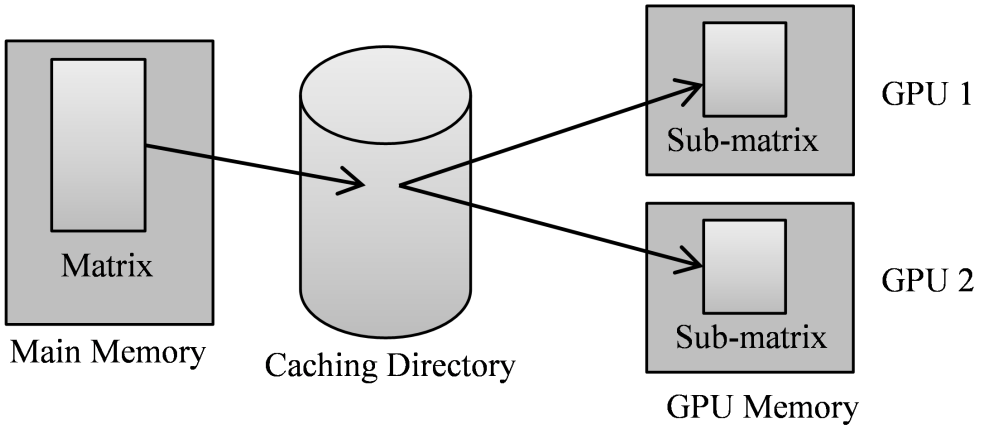

Fig. 5.1. Multi-GPU offloading using the Caching Directory

parallel execution, and overlapping communication with computation. Finally, synchronization is used to prevent data races.

In this sections, these changes will be discussed in details to describe how SemCache++ is built.

5.1 High Level Overview

A SemCache++-enabled library looks, to a programmer, like a typical CPU library. SemCache++ directives (embedded in the library code, not exposed at the interface level) specify what data (matrices) are read and written by the library call. SemCache++ libraries provide multi-GPU implementations by decomposing the computation into subtasks that operate over portions of the data (submatrices). These computations are then distributed across multiple GPUs as part of the library implementation (Section 5.5 discusses a concrete example of how such a library might be implemented). Figure 5.1 shows a high overview of SemCache++ multi-GPU offloading.

SemCache++ can also supports hybrid CPU/GPU execution. Since each submatrix is tracked and executed separately, the CPU can be also exploited in parallel with multiple GPUs to compute part of the result.

SemCache++ manages communication by tracking the locations of the submatrices, identifying whether it is on the CPU, or on one or more GPUs, or shared between the CPU and GPUs. Data is not eagerly communicated, but instead it is only transferred if it is needed by a computation. Because the data remains distributed after a library call completes, when a future library call is issued, the subtasks of that call can be dispatched to appropriate GPUs to reduce communication.

SemCache++ then ensures that the CPU and GPU(s) maintain a consistent view of data by transferring data back from the GPU(s) whenever the CPU requires the data. As in SemCache, SemCache++ determines when a CPU reads or writes data through the use of page protection. Note that while data on the GPUs is tracked at the granularity of decomposed inputs (submatrices), data on the CPU is tracked at the granularity of entire matrices. Thus, SemCache++ uses a two-level directory structure to track data, as described next.

5.2 Cache Design and Structure

SemCache++ uses a directory structure to track the status of data that is used during offloading operations. Figure 5.2 illustrates the design of this directory. As mentioned above, SemCache++ uses a two-level structure: the first level of the directory tracks matrices at the granularity they are used in library calls, while the second level is used to track the submatrices that are distributed across GPUs.

The first level of the directory tracks the matrices that are involved in offloading operations, indexed by the CPU start address (CPU_s) and CPU end address (CPU_e). This first level also records the number of rows in the matrix (n_r), to support efficient memory transfer, as described in Section 5.2.1. Finally, the first level records whether the matrix is valid or invalid on the CPU.

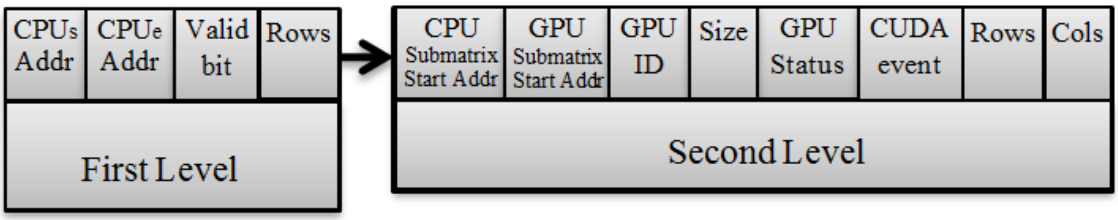

Fig. 5.2. Structure of Caching Directory

SemCache++ assumes that matrices are stored contiguously in memory. When an offloadable method is invoked, the directory can be queried to see if entries exist for the matrix operands of the method. If not, an entry is created. Note that since library methods operate on entire matrices (matrices are only decomposed for distribution across multiple GPUs), lookups into the first level happen at the granularity of matrices.

The first-level entry for a matrix points to a set of *translation records* for the matrix. When a matrix is decomposed into submatrices and distributed across the GPUs, each submatrix is assigned a record in this second level. A translation record serves several purposes. First, it translates between the location of data on the CPU and the corresponding location on the GPUs, facilitating data movement between devices. Second, it keeps track of the coherence state of the data (*i.e.*, where valid copies of the submatrix reside). Finally, it tracks the ready state of the data (*i.e.*, whether the data is available for use by a task). The following sections describe these tasks in more detail.

When a task is launched to execute on a GPU, it uses SemCache++ directives to identify which submatrices are needed for the computation. If the data is already being tracked by the first level, SemCache++ checks the status of the required submatrices in the second level. If the data does not exist on the target GPU, communication is performed.

5.2.1 Translating between CPU and GPU addresses and transferring data

A submatrix is a region of data within the range of a larger matrix. The submatrix may be copied to the GPU as row tiles or column tiles, the translation record stores the start address of the submatrix on the CPU as well as the number of rows and columns. The submatrices are stored contiguously on the GPU, so the translation record tracks the start address of the data on the GPU and the size of the data. Because a submatrix may be replicated on multiple GPUs, the translation record stores the GPU ID and the start address for each GPU the submatrix resides on.

This translation information is used to transfer data back and forth between the CPU and GPUs, as well as for inter-GPU transfers. Inter-GPU transfers are straightforward. If the submatrix is being moved to a GPU that does not currently have a copy of the submatrix, new space is allocated on that GPU and the translation record is updated to reflect the location of that space. When moving a submatrix from the CPU to the GPU, the row and column information stored in the translation record for the submatrix are used to generate a `cublasSetMatrix` call, which provides a single call to transfer an entire tile of a matrix to the specified GPU, allocating memory if necessary. Data tracking is done at the granularity of submatrices. When the CPU requires access to region of data computed on the GPU, only the corresponding submatrix is transferred back using a `cublasGetMatrix` call.

Managing available GPU memory

Using multiple GPUs increases the total available memory space. Kernels that do not fit in a single GPU memory can be executed on multiple GPUs. Although multi-GPU increases the caching space, data might occupy all of the free GPU memory. In such a situation, to allocate new data in the GPU memory, cached data must be freed. To determine which address ranges should be freed, SemCache++ uses least-recently-used (LRU) policy similar to the policy adopted in SemCache as described in Section 4.2.1.

5.3 Coherence Protocols and Instrumentation

5.3.1 Coherence Protocol

SemCache++ uses a modified MSI coherence protocol to track which devices have valid copies of (sub)matrices. The states are tracked through the use of a *valid* bit in the first level entry for a matrix, as well as a *GPU Status* field for each submatrix in the second-level entry. The *CPU valid* bit in the first-level tracks whether or not the matrix is available to the CPU to speedup the lookup process. It is set when all submatrices have CPU only **C** status or shared **S** status. When the *CPU valid* bit is unset, each submatrix can have a different status and the GPU Status field in the second-level entry is used to determine the status as follows:

- **C:** Submatrix exclusive to CPU.
- **S:** Submatrix shared between CPU and GPU(s).
- **G:** Submatrix valid only on GPU(s).

The caching directory records transitions between states in the usual way, triggering communication if necessary. If a task dispatched to a GPU reads a submatrix that is not already on the GPU, then data is transferred (from the CPU if possible, as GPU-GPU communication is often slower), an entry for the submatrix is created, and the status in the second level is set to shared (**S**). SemCache++ allows multiple copies of the same submatrix to exist in the shared status; if another GPU wants the submatrix, then it receives a copy, too. However, like regular caches if a submatrix needs to be written to, all shared copies of the submatrix are discarded and only one submatrix holds a modified state (**G**).

If a matrix is *read* on the CPU while the first-level valid bit is unset, the GPU status is checked in the second level entries. If the status is **G**, submatrices are transferred from the GPU back to the CPU, the valid bit is set, and all submatrices change status to shared (**S** state). If a matrix is *written* on the CPU, then the status of the second level entries becomes **C**, with data transferred back from the GPUs if

necessary. Section 5.3.2 describes the CPU and GPU instrumentation that triggers the state changes in the coherence protocol.

5.3.2 Instrumentation

Instrumenting GPU Reads and Writes

To be able to track the status of GPU data correctly, you need to determine which data is read or written by the GPU. Prior work has used compiler analysis or programmer annotations to determine if the operation is a read or a write [20–23]. Since SemCache++ focuses on libraries, it can use simple directives inserted into the library code to indicate which matrices are read and written by the GPU, as well as which submatrices are needed by tasks dispatched to various GPUs.

Instrumenting CPU Reads and Writes

Similar to SemCache, SemCache++ uses the operating system's virtual memory protection to instrument CPU reads and writes. Page protection can be used to limit access to the CPU data which has been sent to the GPU. For each submatrix that SemCache++ tracks on the CPU, SemCache++ sets page protection flags for all the pages the submatrix spans. The page protection flags are set according to the state of the data structure as follows:

- If the submatrix is in G state, its pages are set to `NO ACCESS`.
- If the submatrix is in S state, its pages are set to `READ ACCESS`.
- If the submatrix is in C state, the pages are set to `READ and WRITE ACCESS`.

If a CPU access triggers a page fault due to write to a no access region or a read only region, SemCache++ looks up the address that caused the page fault in the caching directory. If the translation record is found, it transfers the submatrix back from the GPU and the submatrix status becomes CPU only (C). If a CPU access

triggers a page fault due to a read to a no access region, SemCache++ transfers the submatrix back from the GPU and the submatrix status becomes shared (S).

In order to avoid false sharing between matrices, memory allocation should be page-aligned and padded out to page boundaries.[1] Depending on the matrix decomposition, false sharing might also exit between submatrices which share the same page. If a single submatrix is modified by a GPU, all of the pages the submatrix spans are protected to no access. If this submatrix is invalidated, the other submatrix which shares the same page is conservatively invalidated and both submatrices are transferred back to the CPU.

Page aligned memory allocation can introduce some wasted memory which is negligible for larger data sizes. Usually, it is only profitable to offload medium to large data structures on the GPU to take advantage of the parallelism, where this overhead is minimal (<1% for 400x400 matrix). We note that this overhead is only introduced on the CPU side. On the GPUs, sub-matrices are allocated in variable sizes and do not have to be page aligned.

5.4 Synchronization

To facilitate parallelism, and the overlap of communication and computation, these tasks are launched asynchronously, using CUDA's streams. Moreover, this overlap can occur across library methods, if a second library call uses the same submatrices as the first library call.

Because tasks are launched asynchronously, and from multiple (possibly dependent) library calls, it is important that tasks do not begin to execute until their predecessor tasks complete. SemCache++ takes advantage of CUDA *events*: small kernels that can be launched to streams and act as signals.

Each submatrix has an event handle associated with it, stored in the translation record. Whenever a submatrix is sent to a GPU, or when a submatrix is computed

[1]While page aligning data requires some program modification, identifying allocations to modify is significantly easier than, for example, identifying data accesses to annotate.

(modified) by a task, the operation is performed by dispatching the task to a stream on the target GPU. The event handle associated with the submatrix is then dispatched to the same stream using `cudaEventRecord`. The semantics of streams ensure that this event will not trigger until the previous operation (communication or computation) finishes. In other words, the event will not execute until the submatrix is up-to-date on the target GPU.

Before a communication or computation operation that needs a submatrix is dispatched, SemCache++ must make sure that the submatrix is up-to-date. The submatrix's event handle is dispatched to a stream using `cudaStreamWaitEvent`. This ensures that the operation will not commence until any previous `cudaEventRecord` events associated with the same handle have completed (even if those events were dispatched on different devices). Thus, tasks that require a submatrix will wait until operations that compute or transfer that submatrix complete. Essentially, SemCache++ uses events as full/empty bits, ensuring that consumers of a submatrix wait until producers complete.

Note that task-parallel systems (like StarSs and StarPU) use a complex scheduling runtime to ensure that dependent tasks are executed safely [9, 11]. By taking advantage of CUDA's built-in stream and event constructs, SemCache++ is able to execute tasks in parallel—even across library calls—while relying on the hardware to properly synchronize tasks.

5.5 Adapting a library to use SemCache++

This section describes the process of building a library for multi-GPU offloading using SemCache++. First, we describe how a DGEMM (matrix multiply) call can be decomposed to distribute computations across multiple GPUs. Then we describe how SemCache++ directives can be used to perform automatic data management and synchronization.

5.5.1 Multi-GPU Decomposition and Scheduling

There are multiple parallel algorithms for solving matrix computations on distributed systems (*i.e.* ScaLAPACK [35]). Choosing the right algorithm depends on the underlying network and computing architecture. Many factors can be taken into consideration to determine which algorithm to use like load balancing, optimizing communication and computation-communication ratio. SemCache++ is not tied to a single algorithm, it can be used with any distribution algorithm. It can automatically cache and manage the communication with any type of these algorithms. In our implementation of DGEMM, we adopt a strategy similar to Song *et al.*, which takes advantage of locality to minimize communication [36].

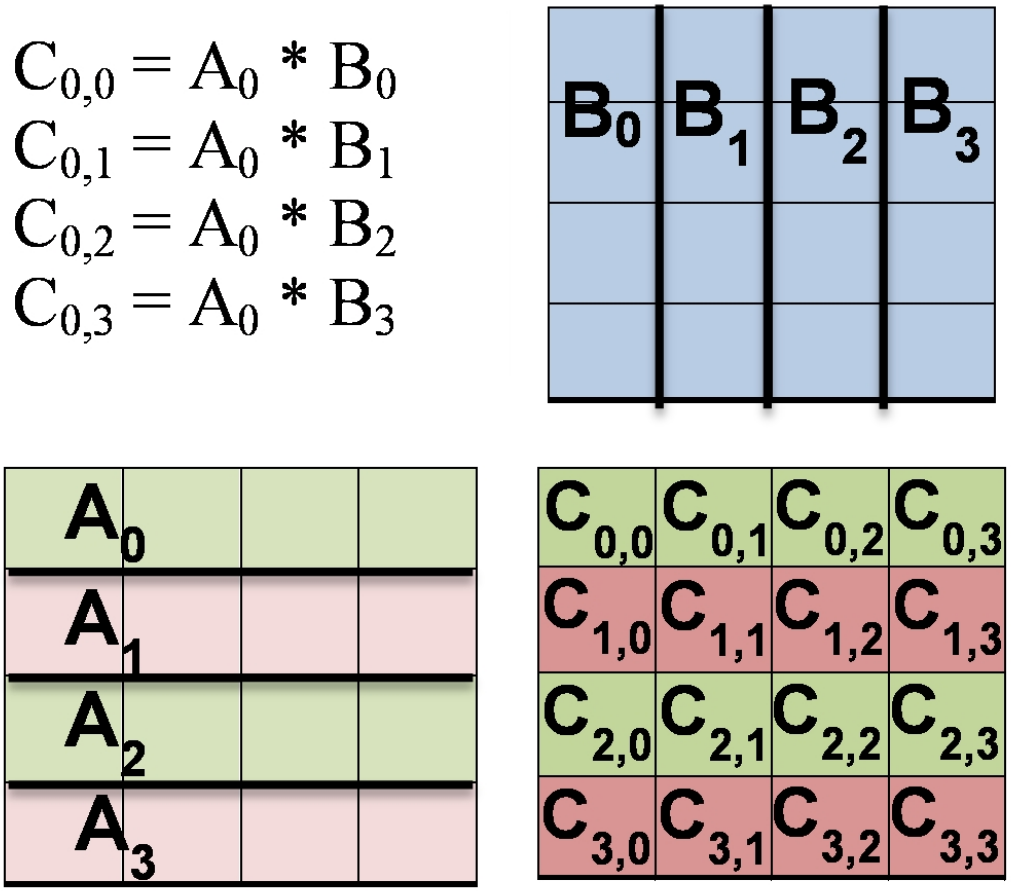

Fig. 5.3. Matrix decomposition

DGEMM calculates $C = \alpha * A * B + \beta * C$. The distribution of the computation across N devices uses a straightforward decomposition. Each matrix is partitioned into N^2 submatrices (an $N \times N$ grid), each of which is tracked separately by SemCache++. For notational convenience, we consider that A's submatrices are grouped into N rows, $A_0, A_1, \ldots, A_{N-1}$, and B's submatrices are grouped into N columns, $B_0, B_1, \ldots, B_{N-1}$. The matrix multiplication is thus broken into N^2 tasks, with a row of A's submatrices being multiplied by a column of B's to produce a single submatrix of C. The decomposition and computation are shown pictorially in Figure 5.3.

As in Song *et al.*, the computation is scheduled by (conceptually) distributing C's submatrices to the N GPUs by dividing the grid of submatrices evenly by rows. Tasks that compute each submatrix of C are then scheduled on the appropriate GPU. Independent tasks are assigned to different streams on the GPU, allowing the computation of one C submatrix to be overlapped with communicating the operands from B for the next task. Figure 5.4 shows how this pipelining can hide communication overheads.

Note that once the computation is completed, each GPU holds a row of A's submatrices and *all* of B. These submatrices remain on the GPUs until another device wants the data. If subsequent calls use the same matrices, mapping tasks to the appropriate GPUs can avoid communication.

CUBLASXT uses a round robin static scheduling policy. Matrices are partitioned based on the specified block size. The block size should be chosen to maximize the overlap between communication and computation. Then the blocks are assigned to the GPUs in a round robin order. Each GPU uses multiple streams. Unlike SemCache++, CUBLASXT does not take locality into consideration which results in excessive communication if the block mapping is not consistent. Consider mapping matrix multiplication on two GPUs. SemCache++ assigns half of the result matrix C to each GPU Fig. 5.5(a) and pipelines the execution on each GPU. CUBLASXT assigns the blocks to the GPUs in a round robin order Fig. 5.5(b). This assignment does not take into consideration data locality as a result it requires sending both

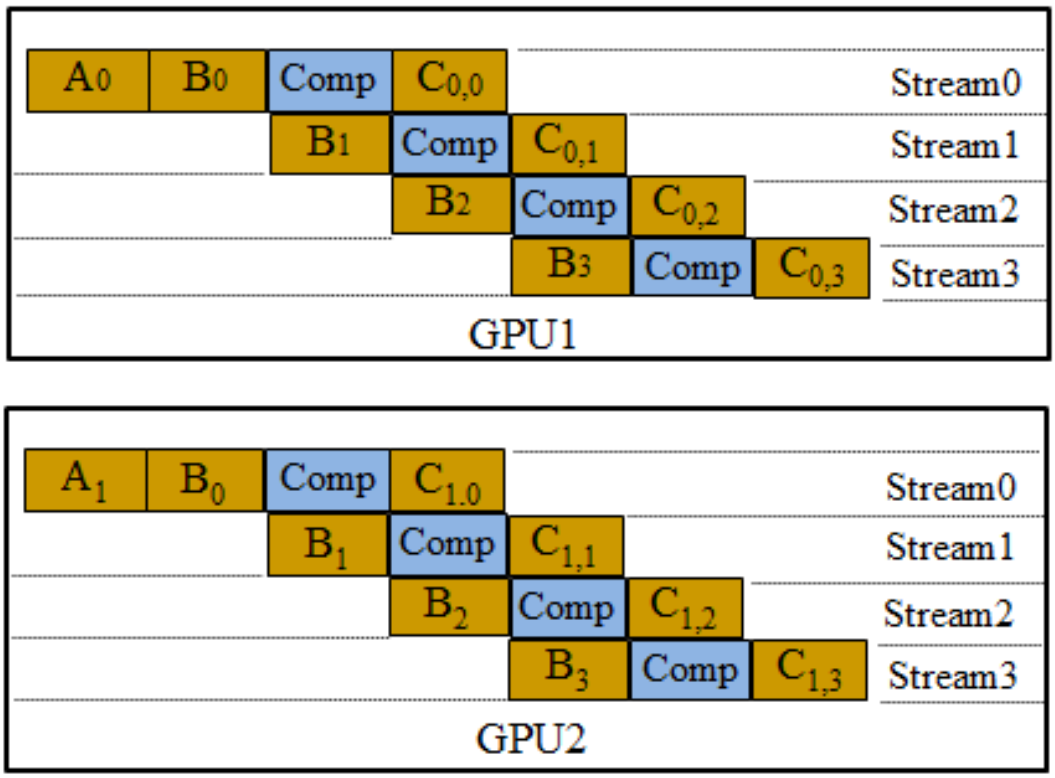

Fig. 5.4. SemCache++ Computation scheduling

matrices (A and B) to each GPU. Where in SemCache++, only half of matrix A is being sent to each GPU.

For compute intensive computations like BLAS level 3, there are multiple levels of overlapping GPU communication and computation. Within a single GPU and across multiple GPUs. To take advantage of overlapping inside a single GPU multiple streams are used. We initialize four streams on each GPU and divide the B matrix into column partitions multiple of four. The partitions are sent asynchronously to the GPU in a pipelined fashion to allow overlapping communication and computation. The size of the partition should be chosen to maximize the overlap. Matrix A is partitioned across multiple GPUs and it can be further divided inside each GPU to maximize the overlap. For example, if the matrix in Figure 5.3 is distributed to four GPUs. Each partition of rows from A is sent to a GPU, with a partition of columns from matrix B. Matrix B partitions are sent to each GPU in a pipelined fashion Figure 5.4. Each GPU will have a tile of the result (e.g. GPU_0 will calculate $C_{0,0}$ to $C_{0,3}$). The result tiles can be received in a pipelined order and overlapped with the computations.

For less compute intensive BLAS routines like level 1 and 2, a simple decomposition can be used. Each matrix can be split by rows into multiple sub-matrices. The number of sub-matrices is equal to the number of GPUs. Each GPU performs part of the computation. Sending the data to the GPU can be pipelined but it has little effect on the performance since the percentage of communication is much higher than the computation.

5.5.2 SemCache++ directives

SemCache++'s API for identifying which computations a task need is similar to the API defined in SemCache, extended to support multiple GPUs. SemCache++ requires that the programmer to specify the number of GPUs using the API method `SemCacheDeviceSelect (numberOfDevices, deviceIds)`. As in SemCache, `readGPU`

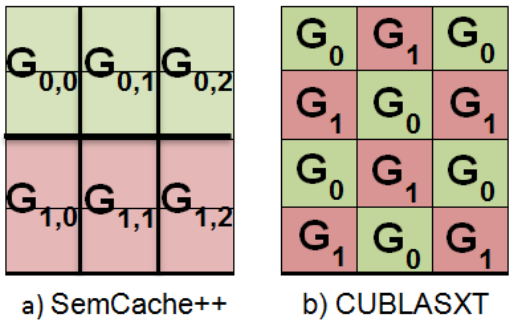

Fig. 5.5. GPU mapping

is used to indicate to SemCache++ that a region of memory (in this case, a submatrix) will be read during a GPU task; the only difference is that in SemCache++, the GPU that will read the submatrix must be identified. Analogously, `writeGPU` is used to indicate that a submatrix was modified by a particular GPU after a task, potentially triggering invalidation of the submatrix on other GPUs or on the CPU.

Because it is common to distribute entire matrices at once, SemCache++ also provides aggregate versions of `readGPU` and `writeGPU` that operate over a whole matrix, decomposing and distributing the matrix across the GPUs. These aggregate functions automatically decompose a matrix into N^2 submatrices and distribute the submatrices by rows or columns to the GPUs. Different decomposition and distribution algorithms exist in SemCache++. Since the decomposition algorithms are not tightly coupled with SemCache++, new algorithms can be easily defined and used. `DecomposeRow` and `DecomposeCol` distribute submatrices by rows and columns, respectively. Figure 5.6 shows how these aggregate functions can be used to manage submatrices for matrix multiply.

Inside the `readGPU` call, a lookup in the caching directory is performed using the start and end address on the CPU and the translation record is returned if found. If data does not exist on the GPU, the matrix is decomposed using the specified decomposition algorithm and sent to multiple GPUs asynchronously as described previously. If data already exist on the device, each submatrix record is inspected as follows: If a

```
1  SemCacheDgemm(TRANSA,TRANSB,M,N,K,ALPHA,
2      A,LDA,B,LDB,BETA,C,LDC)
3  {
4    //A stored in CPU memory range [A, A+(M*K*sizeof(double)))
5    //A will be decomposed to rows and sent to multiple GPUs,
       the submatrix states will be "S"
6    entryA = readGPU(A, M, K, DecomposeRow)
7
8    //B stored in CPU memory range [B, B+(K*N*sizeof(double)))
9    //B will be decomposed to cols and sent to multiple GPUs,
       the submatrix  states will be "S"
10   entryB = readGPU(B, K, N, DecomposeCol)
11
12   //C stored in CPU memory range [C, C+(M*N*sizeof(double)))
13   //If BETA!=0, C will be decomposed to rows and sent to
       multiple GPUs, the submatrix states will be "S"
14   entryC = readGPU(C, M, N, DecomposeRow)
15
16   foreach GPU{
17     foreach stream{
18     //Perform computation on submatrix
19     cublasDgemm(stream,
20       TRANSA,TRANSB,Atiles,Btiles,K,ALPHA,
21       entryA.subRecord.gpu_s,LDA,
22       entryB.subRecord.gpu_s,LDB,BETA,
23       entryC.subRecord.gpu_s,LDC)
24
25     //Issue synchronization event for submatrix C
26     cudaEventRecord(entryC.subRecord.sync_event, stream);
27     }
28   }
29   //C was written by cublasDgemm
30   //Each C submatrix  state will be updated to GPU only "G"
31   writeGPU(C, M, N, DecomposeRow)
32
33 }
```

Fig. 5.6. Pseudocode of SemCache++ matrix multiply (DGEMM)

```
1  //called before a GPU method that reads [s, rows*cols)
2  TranslationRecord readGPU(s, rows, cols, decomposeAlg) {
3    entry = lookup(s, rows, cols);
4
5    if (entry.2level == Null){ //Matrix does not exist on GPU
6      splitAndTransferToMultiGPUs(entry, decomposeAlg);
7      mprotect(s, entry.size,  PROT_READ);
8    }
9    else{
10     foreach submatrix{
11       if (entry.2level.GPU_Status != G || S){ //GPU submatrix
           is not valid
12         TransferToGPU(entry.2level, decomposeAlg);
13       }
14     }
15   }
16   return entry;
17 }
18
19   //called before a CPU method that reads [s, rows*cols)
20   TranslationRecord readCPU(s, rows, cols, decomposeAlg) {
21     entry = lookup(s, rows, cols);
22
23     if (entry.valid == 0){ //CPU data not valid, GPU data
         might be valid
24       foreach submatrix{
25         if(entry.2level.GPU_Status  != C) //if a submatrix is
             not valid on CPU, send if from GPU to CPU
26           transferToCPU(entry.2Level);
27       }
28     }
29     }
30
31     return entry;
32   }
```

Fig. 5.7. Operations to implement coherence protocol

submatrix already resides on the designated GPU, no communication is necessary. If the submatrix is not valid or not on the designated GPU, a synchronization event is issued to ensure that the submatrix is up-to-date, communication is performed and the directory state is updated appropriately. `readGPU` also page-protects the CPU page(s) containing the matrix as read-only, as discussed in Section 5.3.2.

Pinned memory is used to allow overlapping transfers to multiple devices in parallel, it also allows concurrent communication in both direction on Fermi GPUs. Pinned memory allocates page-locked (non-swappable) memory which enables a DMA on the GPU to request transfers to and from the host memory without the involvement of the CPU.

Once all the data is transferred, the individual tasks are executed. Note that all of these kernel invocations occur asynchronously, and hence can be executed simultaneously (there are no dependences in DGEMM). However, because subsequent library calls might use the matrix C, after each task that computes C, `cudaEventRecord` is called on the submatrix's synchronization event so that later tasks wait until the submatrix is computed.

Finally, `writeGPU` changes the state of all C submatrices to modified (**G**). To ensure that CPU accesses to C wait until the computation is complete and then transfer data back from the GPUs, `writeGPU` changes the page protection on C to no access.

5.5.3 Using SemCache++ with complex memory structures

While SemCache++ provides helper methods to aid in distributed matrices across multiple GPUs, not all data structures are amenable to such predictable partitioning and distribution (*e.g.*, 3D matrices, or irregular structures such as trees and graphs). In such cases, SemCache++'s low level API (readGPU and writeGPU) can be used to distribute those data structures by invoking the appropriate methods on each address range for the data structure. It becomes the library writer's responsibility to

appropriately distribute the data structure. For example, to distribute a 3D matrix, SemCache++'s methods can be called individually on the address ranges for each submatrix (multiple transfer calls are needed for non-contiguous matrices) to transfer the matrix and distribute it according to the library writer's distribution algorithm. Performing distribution using the low-level methods obviates the benefits of SemCache++'s distribution functions and multi-level state tracking, but does not preclude the use of its automatic data movement capabilities.

Distributing sparse matrices

As an example of distributing more complex data structures, we have used SemCache++ to provide offloading support for sparse-matrix libraries. Sparse matrices present an interesting challenge to most systems for managing communication between the CPU and the GPU because of their complex layout: a sparse matrix in CSR form has a data array, a row sum array and a column index array. Splitting the matrix between multiple GPUs requires carefully splitting the column index array and recomputing the row sum array.

SemCache++ handles distributing sparse matrices by delegating the distribution to the library implementation. The library can split the sparse matrix representation, recalculating the index arrays for each submatrix as necessary. SemCache++ tracks the individual arrays representing the sparse submatrix as separate submatrices. Recall that SemCache++ tracks submatrices according to a start address, number of columns and number of rows. SemCache++'s tracking of sparse matrices hence works as for any other data structure: if a task requires accessing the sparse matrix, the library issues `readGPU` calls for each of the components of the sparse matrix, and communication is performed as necessary.

This strategy for handling sparse matrices highlights a key advantage of SemCache++'s library-integrated approach to multi-GPU offloading over other approaches. The index arrays that are distributed across GPUs have *different contents* than the

index array that resides on the CPU. Nevertheless, the abstract state of the sparse array is the same: the same data is stored in two different representations, depending on whether it resides on the CPU or on the GPU. SemCache++ establishes a *semantic link* between the two representations, allowing state changes on one device (*e.g.*, changing the contents of the sparse matrix on the CPU) to be reflected on other devices (*e.g.*, by invalidating all of the sparse submatrices on the GPUs). Note that SemCache used the same notion of semantic links to allow, *e.g.*, row-major matrices on the CPU to be represented by column-major matrices on the GPU.

5.6 SemCache++ Experimental Evaluation

To evaluate SemCache++, we built multi-GPU implementations of the library interfaces provided by CUBLAS and CUSPARSE (NVIDIA's single-GPU linear algebra libraries) using SemCache++ directives to manage communication and synchronization. The internal, per-GPU tasks of the SemCache++ implementations were used the single-GPU CUBLAS and CUSPARSE implementations, as described in Section 5.5.2.

We evaluated three benchmarks: First, we looked at a microbenchmark that allowed us to investigate the behavior of SemCache++ as well as other multi-GPU libraries in depth. Next, we looked at two case studies of using SemCache++-enabled libraries to offload computation in two solvers: Jacobi iterative solver (which used dense matrices), and conjugate gradient (which used sparse matrices). The conjugate gradient code is taken directly from NVIDIA's CUDA benchmark suite. We compared the SemCache++ multi-GPU implementations to single-GPU implementations, as well as multi-GPU implementations that used StarPU and CUBLASXT, NVIDIA's tuned multi-GPU library.

We used two platforms to conduct our experiments. Most of our experiments were performed on a server with AMD Opteron Processors and 32GB memory connected via PCIe 2.0 to two NVIDIA Kepler K20 GPUs. These GPUs support compute

capability 3.5 (allowing us to use NVIDIA's Unified Memory as a baseline). The server runs 64-bit Fedora Linux and CUDA version 6. The second platform was used to evaluate offloading to more than two GPUs but it does not support UM. The host has AMD Opteron Processors and 64GB memory connected to eight Tesla M2090 GPUs in an external PCIe expansion chassis (PowerEdge C410x PCIe Expansion Chassis) connected to the CPU using a host interface card (HIC) and iPASS cable. While this platform let us scale to more GPUs, the external configuration of the GPUs meant that communication between the host and the GPUs was much slower. We refer to the first platform as **kepler** and the second as **tesla**.

5.6.1 Microbenchmark performance evaluation

To understand the behavior of SemCache++-enabled applications, we wrote a simple microbenchmark that performs two matrix multiplies and a DAXPY: $D = AB + AC$. Note that the two matrix multiplies share one of their operands (A), and the DAXPY operates on the results of the two multiplications. As a baseline, we used CUDA 6's unified memory along with CUBLAS to implement a communication-optimized single-GPU version of the microbenchmark. We compared this baseline to SemCache++, CUBLASXT and StarPU using one and two GPUs. Unlike SemCache++ and CUBLASXT, StarPU implementation requires rewriting the benchmark using their programming model. CUBLASXT supports multi-GPU computation by carefully overlapping communication with computation. Its performance is dependent on setting the block size for this pipelined schedule. Hence, we evaluate several different block sizes for CUBLASXT on two GPUs. These experiments were conducted on **kepler**.

Figure 5.8 shows the results of the microbenchmark experiment, looking at two different matrix sizes (11K×11K matrices were the largest that could fit on a single GPU for the microbenchmark). We see that even on a single GPU, both SemCache++ and CUBLASXT are faster than the baseline—this is because the baseline does not

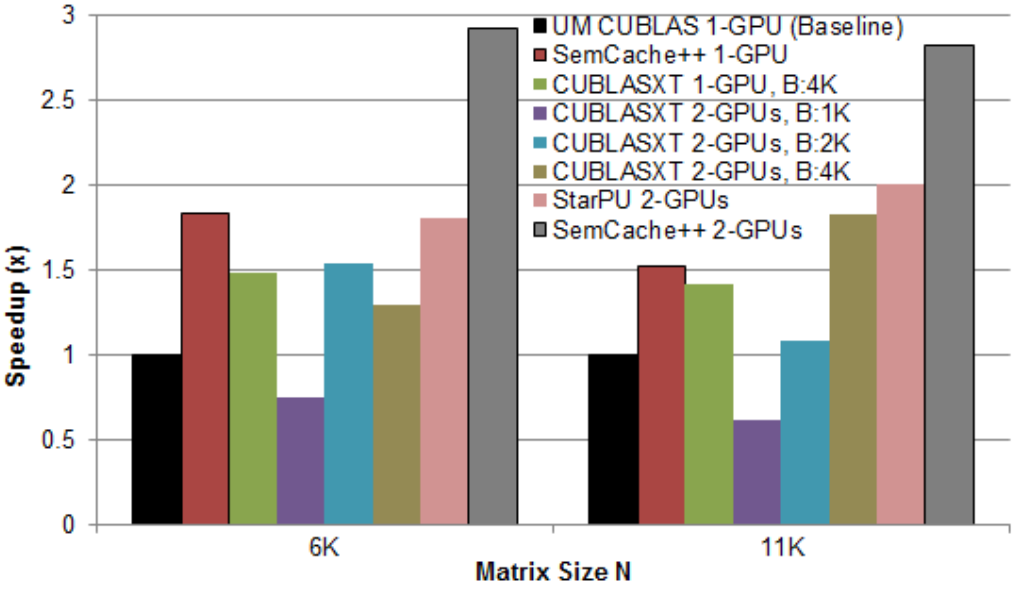

Fig. 5.8. Speedup of microbenchmark for different matrix sizes, normalized to UM CUBLAS 1-GPU)

overlap communication with computation, while both SemCache++ and CUBLASXT do. SemCache++ is faster than CUBLASXT because it is able to minimize communication. The A matrix is cached on both GPUs, as are the results of the DGEMMs. Hence, the DAXPY can be performed with no additional communication. In contrast, CUBLASXT, which does not leave the DGEMM results on the GPUs, must communicate the results of the DGEMMs *back* to the GPUs to perform the DAXPY.

When scaling to two GPUs, we find that SemCache++'s advantage increases: it is nearly 3× faster than the baseline, and 30-50% faster than CUBLASXT and StarPU. StarPU is slightly faster than CUBLASXT because it avoids redundant communication. However, StarPU communication/computation overlapping was limited when synchronization was used to produce correct results which made it slower than SemCache++. We note here a further problem of CUBLASXT's reliance on computation/communication overlap: the optimal block size depends on the input matrix size. In fact, the default block size for CUBLASXT (1K) results in slower performance than a single GPU! This sort of tuning is not necessary for SemCache++,

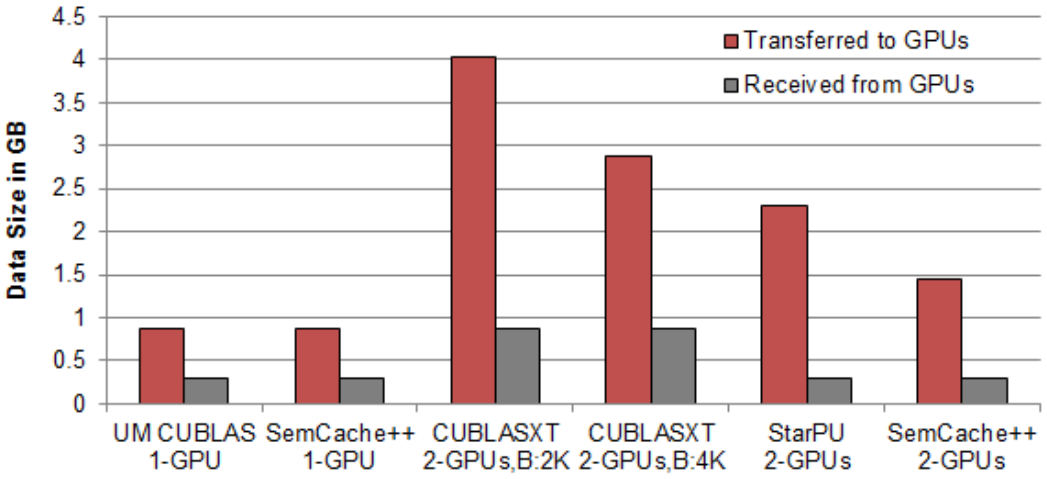

Fig. 5.9. Microbenchmark communication results for size N=6K

which decomposes the matrix into equal blocks regardless of input size (as described in Section 5.5.1). Instead, SemCache++ derives its performance improvement from avoiding redundant communication entirely.

To better understand where SemCache++'s advantages lie, we investigated two possible sources of performance improvement. First, we measured the performance of a *single* matrix multiply using SemCache++'s library and using CUBLASXT. We found that even with the optimal block size, SemCache++'s DGEMM implementation is slightly faster, about 10% for 11K matrices. We speculate this is because SemCache++ uses a simpler matrix distribution than CUBLASXT, resulting in slightly more efficient communication of the matrix operands.

The remainder of SemCache++'s performance improvement comes from optimized communication. Figure 5.9 shows the amount of data transferred to and from the GPU for 6K×6K matrices. SemCache++ transfers significantly less data than CUBLASXT and StarPU. Note that this figure reflects two sources of additional communication. First, StarPU's and CUBLASXT's less efficient matrix decomposition requires more communication to perform a matrix multiplication (this effect is reflected in SemCache++'s 10%-faster DGEMM than CUBLASXT). Second, in

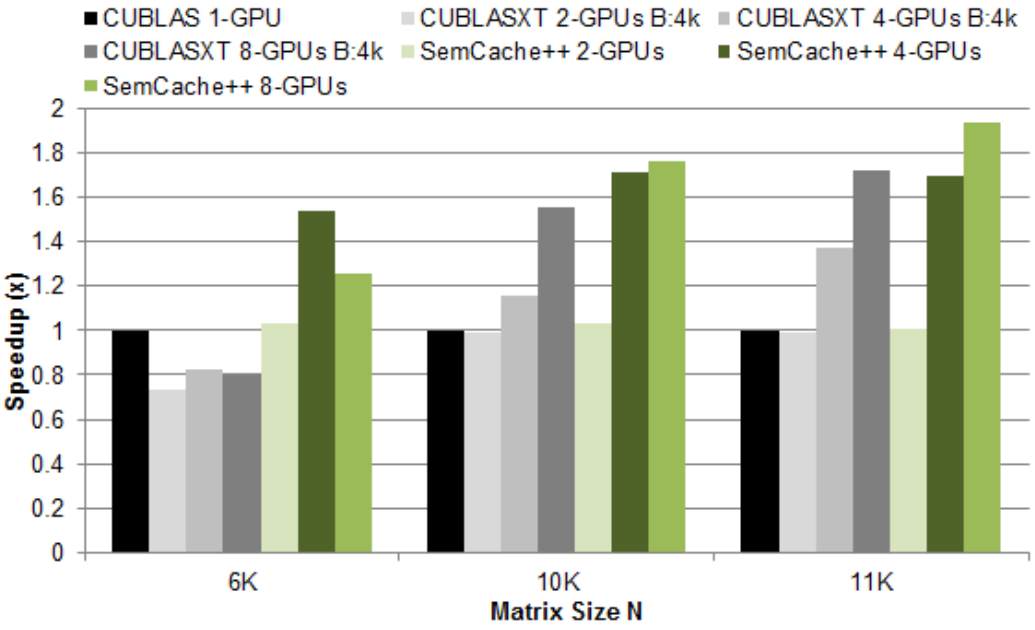

Fig. 5.10. Microbenchmark performance on multiple GPUs for different matrix sizes, speedups with respect to CUBLAS 1-GPU)

the case of CUBLASXT, matrices are re-transferred across library calls, while SemCache++ avoids this communication.

Scalability: Finally, we investigated the scalability of our multi-GPU solution. Figure 5.10 shows the microbenchmark performance on the **tesla** platform, running on up to 8 GPUs. For each matrix size, we show the best-performing CUBLASXT block size. Speedups are limited because communication from the host to the external GPUs is slow, and, unlike with internal GPUs that can take advantage of direct DMA transfers, with external GPUs the bandwidth is divided and hence per-GPU bandwidth decreases with scale. Nevertheless, with the largest matrices, where there is enough computation to amortize the slow communication, we see that SemCache++ is able to provide increasing performance up to 8 GPUs, and is faster than CUBLASXT running on the same number of GPUs.

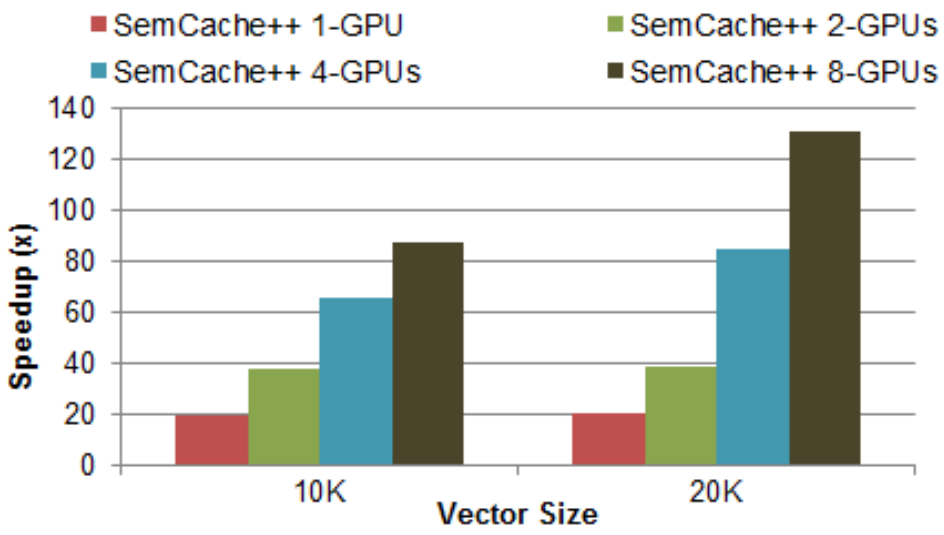

Fig. 5.11. Speedup of Jacobi, normalized to unoptimized CUBLAS

5.6.2 Case Study(I): Jacobi Iterative Solver

The Jacobi iterative solver performs the repeated MvM computation described in the introduction. Figure 5.11 shows Jacobi performance for different vector sizes on the **tesla** platform. Speedups are normalized to the unoptimized CUBLAS implementation. The unoptimized version provides encapsulation; the A matrix is sent to the GPU in every iteration. Running the unoptimized CUBLAS implementation on multiple GPUs did not gain any speedups because communication cost was dominant so the results are not included in the figure. Running Jacobi using SemCache++ on a single GPU achieved 20x speedup because matrix A is cached. For large vector sizes. SemCache++ achieved linear speedups on multiple GPUs. As described in the introduction, each GPU computes part of the vector in each iteration and SemCache++ automatically sends the partial vectors to each GPU using peer to peer transfers. The communication is naturally overlapped, which minimizes the overhead. Note that unlike in our microbenchmark, the ratio of computation to communication is high enough that the slow PCIe bus does not limit scalability.

5.6.3 Case Study(II): Conjugate Gradient

NVIDIA provides two variants of conjugate gradient (CG) in its benchmark suite. In the first, communication is hand-tuned, while in the second, unified memory is used to manage communication. We use the first implementation as the baseline. Because SemCache++ enables drop-in library replacements for BLAS operations, we were able to directly use NVIDIA's unified memory code with SemCache++ to provide multi-GPU offloading.

CG uses CUBLAS and CUSPARSE libraries. Sparse matrix multiplication (SpMV) from the CUSPARSE library uses Compressed Sparse Row (CSR) format for storing matrices. The generated matrix is symmetric tridiagonal. Since the matrix is symmetric any split is balanced, we choose to split it by the number of GPUs. The rows sum is calculated per split. SpMV requires the entire vector for the matrix multiplication. Parts of the vector are computed on each GPU. SemCache++ detects from the caching directory entries that the vector is split on multiple GPUs and it automatically initiates communication to broadcast the vector to all GPUs. Each part of the vector is communicated to the other GPUs using direct GPU to GPU communication. The transfers are overlapped in both directions to double the bandwidth. It is important to note here that this is a general approach in SemCache++. It works for any SpMV kernel, there are no special optimizations done for CG.

Figure 5.12 shows CG performance for different vector sizes. Speedups are normalized to the hand-tuned implementation's execution time on a single GPU. SemCache performance on a single GPU is very close to hand-tuned performance, with 2% overhead due to cache lookups. Unified Memory (UM) is slower than hand-tuned because data transfers from the GPU to the CPU are done at the granularity of pages. Using SemCache++ with 2 GPUs we achieved 1.4x speedup on average over the single GPU hand-tuned baseline version for vector sizes larger than one million. Note that SemCache++ uses NVIDIA's DirectGPU capabilities when transferring data between GPUs. Nevertheless, for smaller matrix sizes, the overhead of peer-to-peer communi-

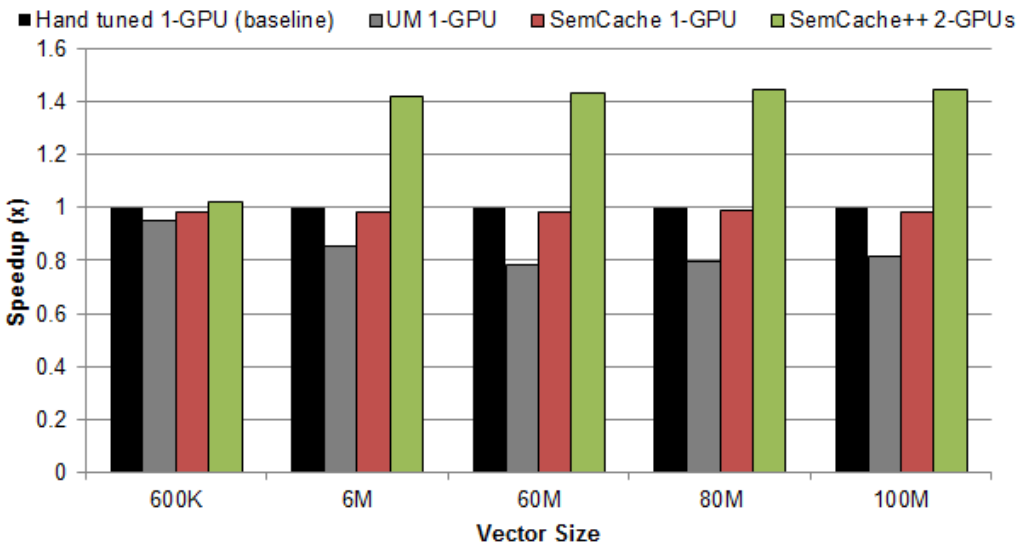

Fig. 5.12. Speedup of CG, normalized to (Hand-tuned 1-GPU)

cation between GPUs limits the performance improvement. These results are consistent with the results of other multi-GPU conjugate gradient solvers [37], which found that the main limiting factor for their speedup was peer-to-peer communication.

Large problem sizes: Multi-GPU execution can not only be used to improve performance; it can also be used to run larger problem sizes than would fit on a single GPU. Splitting the matrix on 2 GPUs enabled us to run CG with sizes double the size that can fit in a single GPU. For example, we were able to run CG for a vector size of 100M on a single GPU using either single-GPU implementation, while with SemCache++ we were able to run double that size (200M) on two GPUs.

6. AUTOMATIC CODE GENERATION AND SYNTHESIS

SemCache takes advantage of library semantics to infer the caching granularity and other required information like input/output matrices and their size. With additional information from the programmer (i.e., annotations), SemCache code can be generated automatically. SemCache automatic code generation tool can save the programmer the effort to manually manage and optimize communication which improves productivity and performance. Trying to optimize communication manually is not feasible for large applications as we explained earlier, that's why a library like SemCache is needed. SemCache can create a mapping between data on the CPU and the GPU to automatically optimize communication.

Some cases where SemCache automatic code generation tool might be needed:

- If a programmer writes a new GPU library to replace a CPU library, he can automatically generate custom SemCache code to manage communication for his library.
- If the programmer needs to apply data transformations for each input/output, he can use this tool to avoid rewriting SemCache interface to integrate these transformations.

Using the automatic code generation tool also allows data transformations to be applied for each input/output. Data transformations creates mappings between different CPU-GPU data representations. For example, non-contiguous structures on the CPU like pointer-based graph representation can be mapped to a contiguous structure on the GPU. Using such transformations in SemCache without using the tool, requires breaking up the modularity of the library calls and prevents the programmer form using SemCache interface because these transformations need to be

integrated into the lookups and the communication operations. Instead, using this tool can easily generate SemCache code with custom transformations inserted for each input without the need to use the library interface or worrying about breaking the modularity of the code. Transformations can be specified at a high level and the tool can integrate it in the code automatically.

This tool can be easily used to map GPU libraries to CPU libraries. If a programmer writes a new GPU kernel to replace a CPU kernel, he can add annotations to the CPU kernel and the SemCache mapping code can be automatically generated. The code is generated once for each library and it could be reused.

The tool requires the following annotations:

- Input matrices and the size (dimensions) for each matrix
- Output matrices and the size (dimensions) for each matrix
- GPU method name

Other optional annotations are:

- Data transformations using a user defined transfer function. Different data transformations can be specified for each input/output separately.
- CPU-GPU matching parameters order. (If not specified, the exact parameters matching is expected)

To illustrate an example for SemCache code generation, we show in figure 6.1 how matrix multiplication subroutine from BLAS library can be annotated. Our tool reads a header file which contains definitions for CPU methods. The file is parsed and the library semantics are inferred from the code and from user annotations. Then the mapping is made between the GPU method and the CPU method. Then SemCache code is generated (figure 6.2) based on the inferred data.

```
1 #pragma matrix inputs:A<M*K>, B<K*N>, C<M*N>; outputs:C<M*N>;
2 #pragma GPUMethod CublasDgemm; ParameterMatch <1-0>
3 DGEMM(TRANSA,TRANSB,M,N,K,ALPHA, A,LDA, B,LDB,BETA, C,LDC)
```

Fig. 6.1. Annotations for DGEMM CPU method

```
1 SemCacheDgemm(int ColumnMajor, char TRANSA, char TRANSB, int M
     , int N, int K, double ALPHA, double* A, int  LDA, double*
     B, int LDB, double BETA, double* C, int LDC){
2   entryA = readGPU(A, A+ (M*K));
3   entryB = readGPU(B, B+ (K*N));
4   entryC = readGPU(C, C+ (M*N));
5   cublasDgemm(TRANSA, TRANSB, M, N, K, ALPHA, entryA.gpu_s,
       LDA, entryB.gpu_s, LDB, BETA, entryC.gpu_s, LDC);
6   writeGPU(C, C+ (M*N));
7 }
```

Fig. 6.2. SemCache automatic generated code for DGEMM

Figure 6.3 shows how transformations can be specified at a high level for the DGEMM method. The input matrices are transformed from row major order to column major order and vise versa for the output matrix. In addition to that, the forward and reverse methods which implement the transformations are specified and highlighted using annotations. Then the tool can integrate the transformations in the generated code automatically (figure 6.4). The generated code unfolds methods readGPU/writeGPU to insert transformations.

```
1  #pragma matrix inputs:A<M*K>, B<K*N>, C<M*N>; outputs:C<M*N>;
2  #pragma transformation
3      forward A: RowToColumn, B: RowToColumn, C: RowToColumn;
4      reverse C: ColumnToRow;
5  #pragma GPUMethod CublasDgemm;
6  DGEMM(TRANSA,TRANSB,M,N,K,ALPHA, A,LDA, B,LDB,BETA, C,LDC)
7
8
9  Void RowToColumn(parameters){
10 //transform data from row major to column major
11 }
12
13
14 Void ColumnToRow(parameters){
15 //transform data from column major to row major
16 }
```

Fig. 6.3. Annotations for DGEMM CPU method with transformations

```
1  SemCacheDgemm(char TRANSA, char TRANSB, int M, int N, int K,
       double ALPHA, double* A, int  LDA, double* B, int LDB,
       double BETA, double* C, int LDC){
2    //unfold method readGPU(A, A+ (M*K))
3    entryA = lookup(A, A+ (M*K));
4    if (entryA.status == C){ //GPU data not current
5      RowToColumn(entryA);
6      transferToGPU(entryA); }
7    //unfold method readGPU(B, B+ (K*N))
8    entryB = lookup(B, B+ (K*N));
9    if (entryB.status == C){ //GPU data not current
10     RowToColumn(entryB);
11     transferToGPU(entryB); }
12   //unfold method readGPU(C, C+ (M*N))
13   entryC = lookup(C, C+ (M*N));
14   if (entryC.status == C){ //GPU data not current
15     RowToColumn(entryC);
16     transferToGPU(entryC); }
17
18   cublasDgemm(TRANSA, TRANSB, M, N, K, ALPHA, entryA.gpu_s,
        LDA, entryB.gpu_s, LDB, BETA, entryC.gpu_s, LDC);
19   //unfold method writeGPU(C, C+ (M*N))
20 # ifdef WRITEBACK
21   invalidateOnCPU(entryC);
22 # else
23   entryC = lookup(C, C+ (M*N));
24   if (entryC.status == G){
25     transferToCPU(entryC);
26     ColumnToRow(entryC); }
27 # endif
28 }
```

Fig. 6.4. SemCache automatic generated code for DGEMM with transformations

7. INTEGRATING SEMCACHE WITH TRILINOS

The Trilinos Project [12] is an effort to facilitate the design, development, integration and ongoing support of mathematical software libraries. It provides parallel solver algorithms and libraries within an object oriented software framework for the solution of large-scale, complex multiphysics engineering and scientific applications. Trilinos uses a two-level software structure designed around collections of packages. Packages exist underneath the Trilinos top level, which provides a common look-and-feel, including configuration, documentation, licensing, and bug-tracking. Trilinos is an open source platform written in C++.

Kokkos [38] is a package in Trilinos for manycore performance portability. Kokkos provides an abstraction of the underlying hardware. It enables performance portable user code which runs on CPUs or GPUs if that code is implemented with Kokkos multidimensional arrays and parallel execution capabilities. It supports parallelism using MPI, CUDA and threads.

Kokkos provides a high level API to allow the programmer to manage communication between the CPU and the GPU. The API requires the programmer to manually specify transfers between different devices. As discussed in the introduction, manual transfers are prune to errors and may result in extra communication. To address this problem, SemCache is integrated with Kokkos to automatically control communication. In this section we discuss the Kokkos package and the integration details.

7.1 Kokkos Package

Kokkos supports a high performance computing (HPC) environment, a network of compute nodes where each compute node contains one or more manycore devices. An HPC application has two levels of parallelism: (1) distributed memory parallelism

typically supported through a Message Passing Interface (MPI) library and (2) thread level parallelism on the manycore device.

Kokkos implements its own device-aware multidimensional array. The array layout is optimized at compile-time for memory accesses. The arrays are implemented by the C++ View template class. Each device has its own view of the data. Since different data layouts may exist between the GPU view and the CPU view, Kokkos uses a HostMirror view in the host memory space to store the devices layout.

```
1  typedef View<double**[8][3],Device> my_array_type;
2
3  my_array_type a("a",N,M); // Allocate on Device
4
5  //  my_array_type::HostMirror  defines an array
6  // in host space with a layout mirroring
7  //   my_array_type  . If the device != host then
8  //   create_mirror_view  allocates a compatible
9  // array, otherwise the input view is returned.
10 my_array_type::HostMirror
11 host_a = create_mirror_view( a );
12
13 // Deep copy to a mirror does not require remap.
14 // If a == a_host deep copy is skipped.
15 deep_copy( a , host_a ); // Copy device <- host
16 deep_copy( host_a , a ); // Copy host <- device
```

Fig. 7.1. Deep copy performance penalties associated with remapping array layouts are avoided by using HostMirror views that have the same layout as a device view but with member values residing in the host space.

Kokkos DualView container class to manage data structures which exist both on Host and Device. The class contains both a GPU view and a CPU view of the data. The Kokkos Vector class which is used to initialize arrays inherits the DualView class.

7.2 Kokkos Integration with SemCache

Kokkos DualView class can be accessed directly from vectors to manage communication. To enhance the properties of Kokkos DualView class, it extends SemCache class which automatically handles communication. The integration code for SemCache with Kokkos is listed in appendix A.

7.2.1 Allocation

Allocation in Kokkos HostSpace is modified to page aligned using *valloc* and padded to prevent false sharing as show in Figure 7.2.

```
1 #define PageMask (PageSize - 1LLU)
2 #define PageCeiling(ArraySize) ((ArraySize + PageSize - 1) & ~
     PageMask)
3 ptr = valloc( PageCeiling(scalar_size * count_alloc) );
```

Fig. 7.2. SemCache Allocation in Kokkos HostSpace

7.2.2 Using SemCache with Kokkos

Since SemCache extends the Vector properties, SemCache directives can be accessed directly from the vector. Before using a vector in GPU computations, readGPU() is called. If the vector stores the output result, writeGPU() is called after the computation. Figure 7.3 shows how SemCache directives are used in a vector addition example in Kokkos.

7.3 Experimental Results

MiniFE is a hybrid parallel (MPI+X) finite element application that constructs a linear system of equations for a 3D heat diffusion problem and performs 200 iterations

```
1  template<typename VectorType>
2  void
3    waxpby(typename VectorType::ScalarType alpha, VectorType& x,
4           typename VectorType::ScalarType beta, VectorType& y,
5           VectorType& w)
6  {
7    int size = y.local_size<x.local_size?y.local_size:x.
       local_size;
8
9    w.coefs.readGPU();
10   x.coefs.readGPU();
11   y.coefs.readGPU();
12
13   if(alpha==1.0)
14     Kokkos::V_Add(w.coefs.d_view,x.coefs.d_view,beta,y.coefs.
         d_view,size);
15   else
16     Kokkos::V_Add(w.coefs.d_view,alpha,x.coefs.d_view,beta,y.
         coefs.d_view,size);
17   device_device_type::fence();
18
19   w.coefs.writeGPU();
20 }
```

Fig. 7.3. SemCache Use in Kokkos

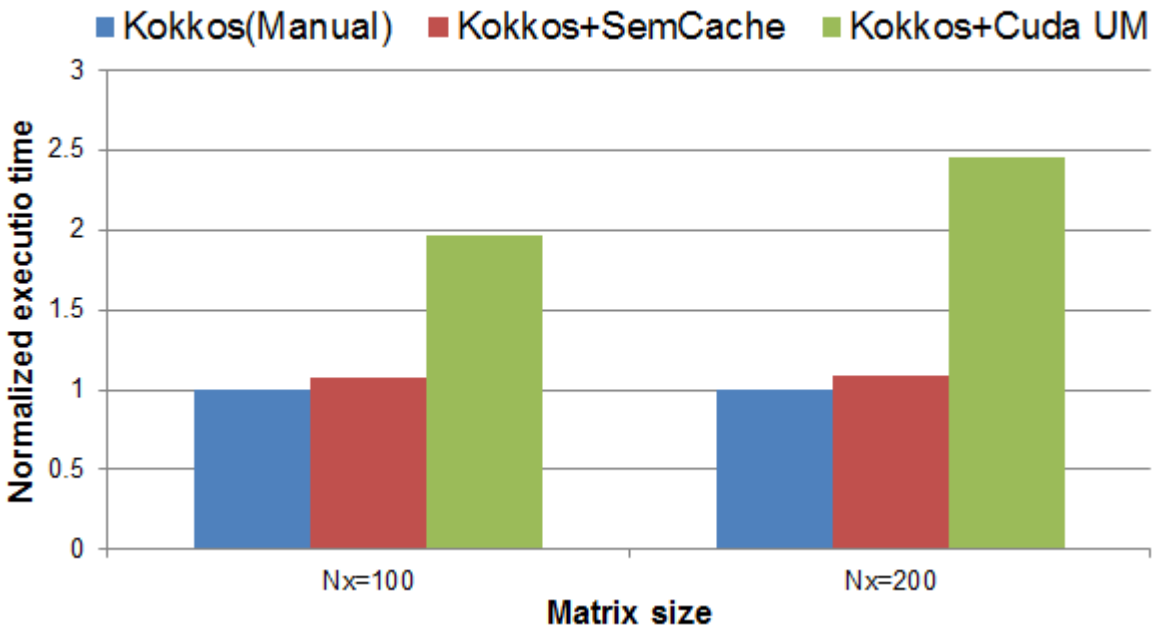

Fig. 7.4. Normalized execution time of CG

of a conjugate gradient (CG) solver on that linear system. It is designed to capture a number of important characteristics of implicit parallel finite element codes. MiniFE has been implemented in various programming models some of which are available at mantevo.org.

Figure 7.4 shows the normalized execution time for conjugate gradient using two different matrix sizes. We compare the performance of miniFE-Kokkos with miniFE-Kokkos enhanced with SemCache and miniFE-Kokkos using Cuda Unified Memory. The results are normalized to miniFE-Kokkos manual communication handling using the Dual View Class as described in the previous section. The results show that the overhead of SemCache is negligible (less than 5%), while the overhead for Cuda Unified Memory is 50% or more. The slow down of Cuda UM is a result of tracking and transferring data at page granularity. SemCache does not suffer from this problem because it tracks and transfers data at the granularity of a matrix.

8. CONCLUSIONS

GPU libraries have made it easy to improve application performance by offloading computation to the GPU. However, using such libraries still introduces the complexity of managing explicit data movement. Unfortunately, when using these libraries with complex applications with multiple levels of abstraction, it is very difficult to reason about how multiple kernel invocations interact with one another, and hence avoid redundant communication. This task is even harder in multi-GPU libraries since they hide the complexity of decomposing data, distributing computations and handling communication manually inside library calls. Such encapsulation prevents the reuse of the data between successive kernel invocations resulting in redundant communication.

In this thesis, we introduced SemCache, a semantics-driven caching technique that can automatically manage and optimize CPU-GPU communication. SemCache tunes its granularity based on the semantics of the GPU libraries in an application. SemCache++ extends SemCache to support offloading to multiple GPU. SemCache++ is used to build the first multi-GPU drop-in replacement library that (a) uses the virtual memory to automatically manage and optimize multi-GPU communication and (b) requires no program rewriting or annotations. Our caching technique is efficient; it only tracks matrices/sub-matrices instead of tracking every memory access at fine granularity. We applied SemCache to Basic Linear Algebra Subprograms (BLAS) [2] library to provide a GPU drop-in replacement library.

Experimental results show that our system can dramatically reduce redundant communication for real-world computational science application and deliver significant performance improvements, beating GPU-based library implementations like CULA [5], CUBLAS and CUBLASXT [6].

LIST OF REFERENCES

LIST OF REFERENCES

[1] N. AlSaber and M. Kulkarni, "Semcache: Semantics-aware caching for efficient gpu offloading," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), pp. 421–432, ACM, 2013.

[2] BLAS, "Basic linear algebra subprograms." `http://www.netlib.org/blas/`.

[3] N. Al-Saber and M. Kulkarni, "Semcache++: Semantics-aware caching for efficient multi-gpu offloading," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, (New York, NY, USA), pp. 255–256, ACM, 2015.

[4] P. D. S. Tomov, R. Nath and J. Dongarra., "MAGMA version 0.2 users' guide." `http://icl.eecs.utk.edu/magma/`, 2009.

[5] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," *SPIE Defense and Security Symposium (DSS)*, pp. 770502–770502–7, 2010.

[6] NVIDIA, "CUDA toolkit 4.0 CUBLAS library." `http://docs.nvidia.com/cuda/cublas/index.html`.

[7] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," *SIGPLAN Not.*, vol. 44, pp. 121–130, Feb. 2009.

[8] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK users' guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[9] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par 2009 Parallel Processing* (H. Sips, D. Epema, and H.-X. Lin, eds.), vol. 5704 of *Lecture Notes in Computer Science*, pp. 863–874, Springer Berlin Heidelberg, 2009.

[10] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "Ptask: Operating system abstractions to manage gpus as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 233–248, ACM, 2011.

[11] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the StarSs programming model for platforms with multiple GPUs," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pp. 851–862, 2009.

[12] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[13] NVIDIA, "Cuda." `https://developer.nvidia.com/cuda-toolkit`.

[14] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 45–55, 2009.

[15] C.-Y. Shei, P. Ratnalikar, and A. Chauhan, "Automating GPU computing in MATLAB," in *Proceedings of the international conference on Supercomputing*, ICS '11, pp. 245–254, 2011.

[16] J. A. Pienaar, A. Raghunathan, and S. Chakradhar, "MDR: performance model driven runtime for heterogeneous parallel platforms," in *Proceedings of the international conference on Supercomputing*, ICS '11, pp. 225–234, 2011.

[17] NVIDIA, "Cuda toolkit 6.0 cublasxt library." `https://developer.nvidia.com/cublasxt`, 2014.

[18] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," *SIGPLAN Not.*, vol. 43, pp. 287–296, Mar. 2008.

[19] R. Vasudevan, S. S. Vadhiyar, and L. V. Kalé, "G-charm: an adaptive runtime system for message-driven parallel applications on hybrid systems," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, (New York, NY, USA), pp. 349–358, ACM, 2013.

[20] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," *SIGPLAN Not.*, vol. 47, pp. 142–151, June 2011.

[21] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 33–42, ACM, 2012.

[22] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for CPU-GPU architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pp. 165–174, 2012.

[23] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pp. 347–358, 2010.

[24] C. Amza, A. L. Cox, H. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, pp. 18–28, 1996.

[25] B. Nitzberg and V. Lo, "Distributed shared memory: a survey of issues and algorithms," *Computer*, vol. 24, pp. 52 –60, aug. 1991.

[26] N. Agarwal, D. Nellans, M. OConnor, S. W. Keckler, and T. F. Wenisch, "Unlocking bandwidth for gpus in cc-numa systems,"

[27] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in opencl for multiple gpus," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), pp. 277–288, ACM, 2011.

[28] S. Schaetz and M. Uecker, "A multi-gpu programming library for real-time applications," in *Algorithms and Architectures for Parallel Processing* (Y. Xiang, I. Stojmenovic, B. Apduhan, G. Wang, K. Nakano, and A. Zomaya, eds.), vol. 7439 of *Lecture Notes in Computer Science*, pp. 114–128, Springer Berlin Heidelberg, 2012.

[29] Y.-P. You, H.-J. Wu, Y.-N. Tsai, and Y.-T. Chao, "Virtcl: A framework for opencl device abstraction and management," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, (New York, NY, USA), pp. 161–172, ACM, 2015.

[30] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, pp. 589 –604, july 2005.

[31] M. Gonzàlez, N. Vujic, X. Martorell, E. Ayguadé, A. E. Eichenberger, T. Chen, Z. Sura, T. Zhang, K. O'Brien, and K. O'Brien, "Hybrid access-specific software cache techniques for the Cell BE architecture," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 292–302, 2008.

[32] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pp. 31:1–31:11, 2008.

[33] C. Liu, M. H. Jamal, M. Kulkarni, A. Prakash, and V. Pai, "Exploiting domain knowledge to optimize parallel computational mechanics codes," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), pp. 25–36, ACM, 2013.

[34] A. Prakash and K. D. Hjelmstad, "A FETI-based multi-time-step coupling method for Newmark schemes in structural dynamics," *International Journal for Numerical Methods in Engineering*, vol. 61, no. 13, pp. 2183–2204, 2004.

[35] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert, "A proposal for a heterogeneous cluster scalapack (dense linear solvers)," *Computers, IEEE Transactions on*, vol. 50, pp. 1052–1070, Oct 2001.

[36] F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), pp. 365–376, ACM, 2012.

[37] M. Verschoor and A. C. Jalba, "Analysis and performance estimation of the conjugate gradient method on multiple {GPUs}," *Parallel Computing*, vol. 38, no. 1011, pp. 552 – 575, 2012.

[38] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

APPENDICES

A. SEMCACHE INTEGRATION CODE WITH KOKKOS

```
1  struct TranslationRecord{
2    void* HostStartAddress;
3    void* HostEndAddress;
4    void* DeviceStartAddress;
5    int Size;
6    int Size_Padded;
7    int DataType;
8    char Status;
9    };
10 std::list<TranslationRecord*> cacheDirList;
11
12 class SemCache{
13 public:
14   TranslationRecord mainEntry;
15
16   void SemCacheSet(void* _HostStartAddress, void*
        _HostEndAddress, void* _DeviceStartAddress, int _Size,
        int _DataType){
17     mainEntry.HostStartAddress = _HostStartAddress;
18     mainEntry.HostEndAddress = _HostEndAddress;
19     mainEntry.DeviceStartAddress = _DeviceStartAddress;
20     mainEntry.Size = _Size;
21     mainEntry.DataType = _DataType;
22     mainEntry.Size_Padded = PageCeiling(_DataType * _Size);//
          page aligned padding
23   }
24
25   static void invalidateOnGPU(TranslationRecord* entry) {
26     (*entry).Status = 'C';
27   }
28   static void invalidateOnCPU(TranslationRecord* entry) {
29     (*entry).Status = 'G';
```

```
30  }
31
32  //called after invoking a GPU method that writes [s, e)
33  void writeGPU() {
34    if(mainEntry.HostStartAddress == 0)
35      return;
36
37    TranslationRecord* entry = lookupAndAdd();
38    invalidateOnCPU(entry);
39    improtect(mainEntry.HostStartAddress , mainEntry.
         Size_Padded ,  PROT_NONE);
40  }
41
42  //called before invoking a GPU method that reads [s, e)
43   void readGPU() {
44    if(mainEntry.HostStartAddress == 0)
45      return;
46
47    TranslationRecord* entry = lookupAndAdd();
48    if ((*entry).Status == 'C'){ //data not current on GPU
49
50      Kokkos::Impl::DeepCopy<CudaSpace,HostSpace>::DeepCopy(
           mainEntry.DeviceStartAddress, mainEntry.
           HostStartAddress,  mainEntry.Size * mainEntry.
           DataType ); //transferToGPU
51        DeepCopyTimer= gettimer() - DeepCopyTimer;
52      std::cout << " DeepCopy time:" << DeepCopyTimer << "\n";
53
54      //std::cout << "mprotect range " << &h_view(0) << "-" <<
           &h_view(0)+(*entry).Size_Padded  << " size:" << (*
           entry).Size_Padded * (*entry).DataType << "\n";
55      mprotect(mainEntry.HostStartAddress , mainEntry.
           Size_Padded,  PROT_READ);
56
57      (*entry).Status = 'S';
58    }
59  }
60
```

```
61    //execute after writing address range [s, e) on CPU
62    static void  writeCPU(TranslationRecord* entry) {
63      //TranslationRecord* entry = lookupAndAdd();
64      if ((*entry).Status == 'G'){ //data not current on CPU
65        Kokkos::Impl::DeepCopy<HostSpace,CudaSpace>::DeepCopy(
              entry->HostStartAddress, entry->DeviceStartAddress,
              entry->Size * entry->DataType );
66      }
67      invalidateOnGPU(entry);
68    }
69
70    //execute before reading address range [s, e) on CPU
71    static void readCPU(TranslationRecord* entry) {
72      //TranslationRecord* entry = lookupAndAdd();
73      if ((*entry).Status == 'G'){ //data not current on CPU
74        Kokkos::Impl::DeepCopy<HostSpace,CudaSpace>::DeepCopy(
              entry->HostStartAddress, entry->DeviceStartAddress,
              entry->Size * entry->DataType );
75
76        (*entry).Status = 'S';
77      }
78      mprotect(entry->HostStartAddress , entry->Size_Padded ,
           PROT_READ);
79    }
80
81  TranslationRecord* lookupAndAdd(){
82      for (std::list<TranslationRecord*>::iterator it=
           cacheDirList.begin(); it != cacheDirList.end(); ++it){
83        if( (*it)->HostStartAddress ==  mainEntry.
              HostStartAddress){// && (cacheDir[x].HostEndAddress
              >= HostEndAddress) ){
84         //std::cout << "=list start addr " << (*it)->
              HostStartAddress  << "\n";
85          TranslationRecord* tr = *it;
86          return tr;
87        }
88      }
89
```

```
90      mainEntry.Status = 'C';
91      cacheDirList.push_back(&mainEntry);
92      return &mainEntry;
93    }
94
95    static TranslationRecord existsInCacheDirRange(void* CPUptr)
96    {
97      for (std::list<TranslationRecord*>::iterator it=
           cacheDirList.begin(); it != cacheDirList.end(); ++it){
98        if( ((*it)->HostStartAddress <=  CPUptr) && ((*it)->
             HostEndAddress >= CPUptr) ){
99          TranslationRecord* tr = *it;
100         return *tr;
101       }
102     }
103
104     std::cout<< "SIG FAULT Address not found exiting \n";
105     exit(EXIT_FAILURE);
106   }
107
108   static void remove(void* CPUptr){
109     for (std::list<TranslationRecord*>::iterator it=
           cacheDirList.begin(); it != cacheDirList.end(); ++it){
110       if( (*it)->HostStartAddress ==  CPUptr){
111         std::cout << "= remove list start addr " << (*it)->
               HostStartAddress  << "\n";
112         cacheDirList.erase(it);
113       }
114     }
115   }
116   static void handler(int sig, siginfo_t *si, void *uap)
117   {
118     ucontext_t *context =  (ucontext_t *) uap;
119     int write_fault = context->uc_mcontext.gregs[REG_ERR];
120
121     TranslationRecord translationRecord =
           existsInCacheDirRange(si->si_addr);
122
```

```
123     if(mprotect(translationRecord.HostStartAddress ,
           translationRecord.Size_Padded,  PROT_READ|PROT_WRITE)
           != 0){
124       std::cout << "**mprotect free failed: " << "\n";
125       exit(EXIT_FAILURE);
126     }
127
128     if (write_fault == 4) //None -> Needs Read Access
129     {
130       readCPU(&translationRecord);
131     }
132     else if (write_fault == 6) //None -> Needs Write Access
133     {
134       writeCPU(&translationRecord);
135     }
136     else if (write_fault == 7) //ReadOnly -> Needs Write
           Access
137     {
138       invalidateOnGPU(&translationRecord);//Status = 'C';
139     }
140   }
141
142   static void Init() //InitHandler
143   {
144     int pagesize = sysconf(_SC_PAGE_SIZE);
145     struct sigaction sa;
146     sa.sa_flags = SA_SIGINFO;
147     sigemptyset(&sa.sa_mask);
148     sa.sa_sigaction = handler;
149     if (sigaction(SIGSEGV, &sa, NULL) == -1){
150       perror("sigaction");
151       exit(EXIT_FAILURE);
152     }
153   }
154
155 };
```

VITA

VITA

Nabeel graduated in 2006 with a Bachelor in Computer Engineering from the University of Jordan. In 2008, he received his masters degree in Computer Engineering from New Jersey Institute of Technology. Nabeel worked at the industry for two years as a senior software engineer. Then, he continued his Ph.D. in computer engineering at Purdue University. His research focused on improving the performance of scientific applications using heterogeneous platforms (GPGPU computing). The project was funded by the Department of Energy with the collaboration of Sandia Labs. His advisor is Professor Milind Kulkarni. Nabeel also worked as a teaching assistant at Purdue. He received excellence in teaching award: "Estus H. and Vashti L. Magoon Award" in May 2012. In 2015, Nabeel was awarded his Ph.D. degree from Purdue University. Nabeel plans to join Qualcomm after his graduation.